Developing Imperfect Software

How to Prepare for Production Pipeline Failure

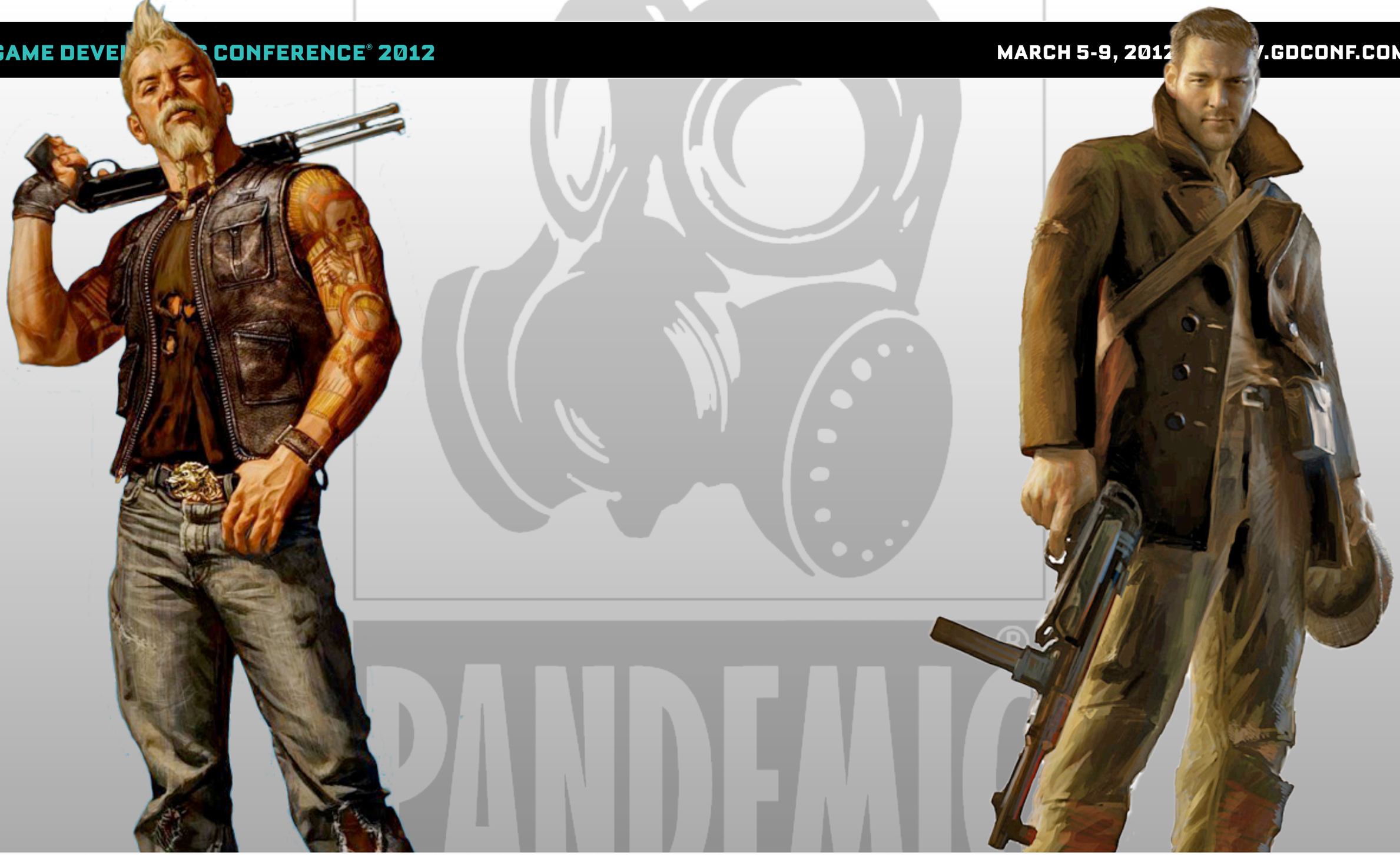
Ron Pieket – Senior Engineer

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9
2012

Thank you all for coming. I'm glad that you decided to stick around for this very last session of GDC 2012. Please silence your cell phones.

My name is Ron Pieket. I'm a senior engineer at Insomniac Games, where I primarily work on productivity tools and pipeline.

Before that I was at Pandemic Studios for many years, where I worked on Mercenaries and Mercenaries 2. After Mercenaries 2 was completed, I joined the Saboteur team, to help them finish the game.



It was after that switch that I really started to think about production pipeline issues. And how we, as engineers, can help our content team experience less downtime, be more productive, and make the game more awesome.

When I joined The Saboteur, the team was already in crunch mode. People were working long hours, weekends, some hardly ever went home. Kids would show up on weekends because they never got to see their daddy – I'm sure this is a familiar scenario. (Cue violins)

But what struck me was that although these dedicated people were working all these extra hours, for a good portion of the day they were sitting on their hands. They were sitting on their hands because the build was broken.

I will be using this term quite a bit throughout this talk. I want to make clear that when I say that “the build is broken”, I don't mean that the code won't compile. That's a simple issue. What I mean by breakage is that one moment you are fighting nazi's, aliens, or zombies, and the next moment you are not.

[CLICK] A assertion fails, the game freezes up, the screen goes black… or maybe you just lose an important functionality in the game, the AI stops running or the hero falls through the terrain.

And in particular, I will focus on the kind of breakage that is caused by programming errors, that then causes the artists and designers to not be able to do their work. There is a whole other category of breakage that is caused by errors in the content data, but the topic of this talk is already quite broad.

I will be using this term quite a bit throughout this talk. I want to make clear that when I say that “the build is broken”, I don't mean that the code won't compile. That's a simple issue. What I mean by breakage is that one moment you are fighting nazi's, aliens, or zombies, and the next moment you are not.

[CLICK] A assertion fails, the game freezes up, the screen goes black… or maybe you just lose an important functionality in the game, the AI stops running or the hero falls through the terrain.

And in particular, I will focus on the kind of breakage that is caused by programming errors, that then causes the artists and designers to not be able to do their work. There is a whole other category of breakage that is caused by errors in the content data, but the topic of this talk is already quite broad.

Production downtime

And breakage affects us greatly. You see, we are not just DEVELOPING the game. We are also USING it. We, the team, are the first users, the first consumers of the game. We rely on it to do our job. And when the build fails, people end up sitting on their hands.

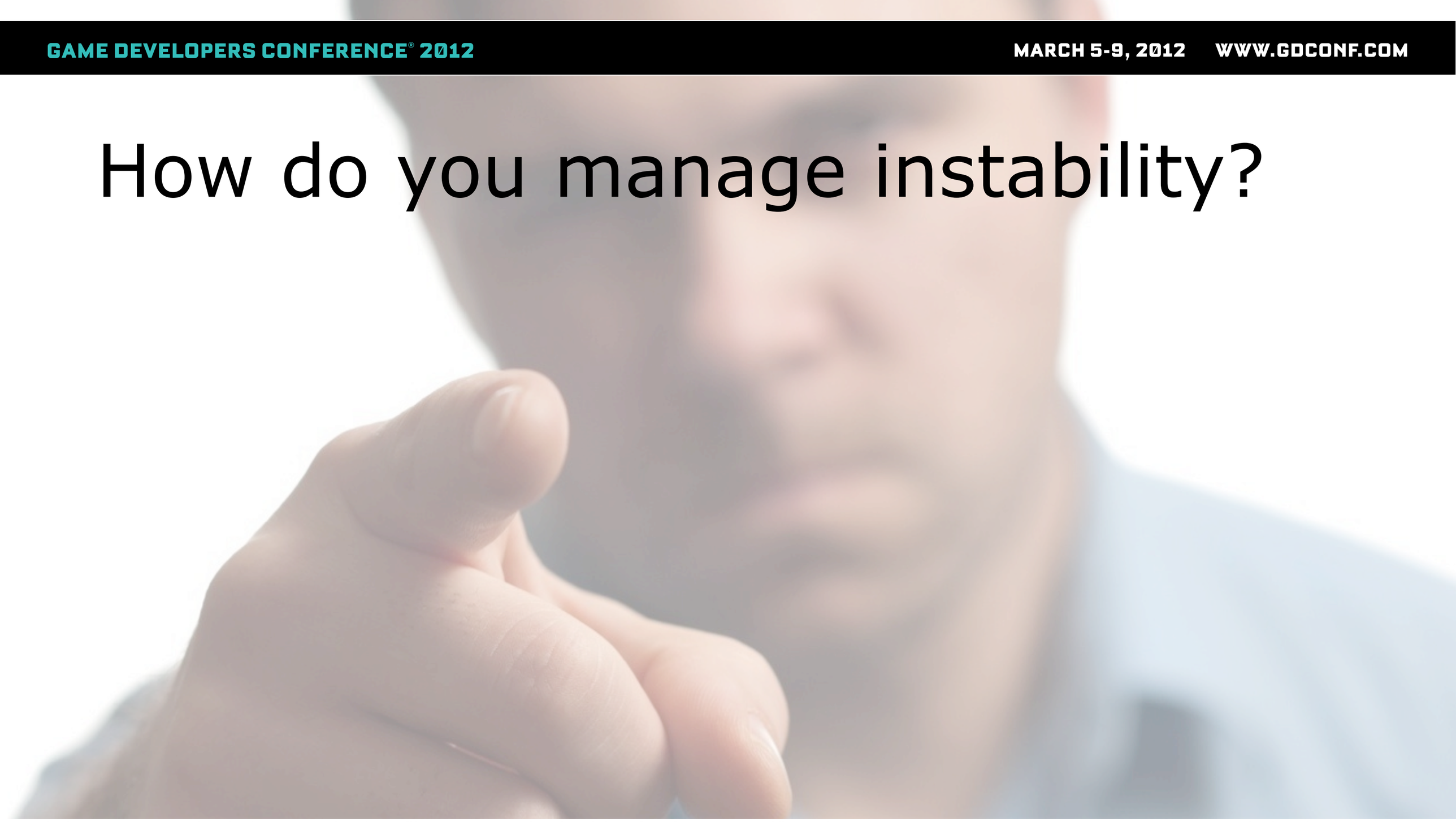
And of course we know that. When the build is broken, engineers rush to find and fix the problem, get a patch out to the team with great haste, because people are sitting still.

Production downtime

- Developers use the same software that they develop
- Software that is in development is inherently unstable

And breakage affects us greatly. You see, we are not just DEVELOPING the game. We are also USING it. We, the team, are the first users, the first consumers of the game. We rely on it to do our job. And when the build fails, people end up sitting on their hands.

And of course we know that. When the build is broken, engineers rush to find and fix the problem, get a patch out to the team with great haste, because people are sitting still.



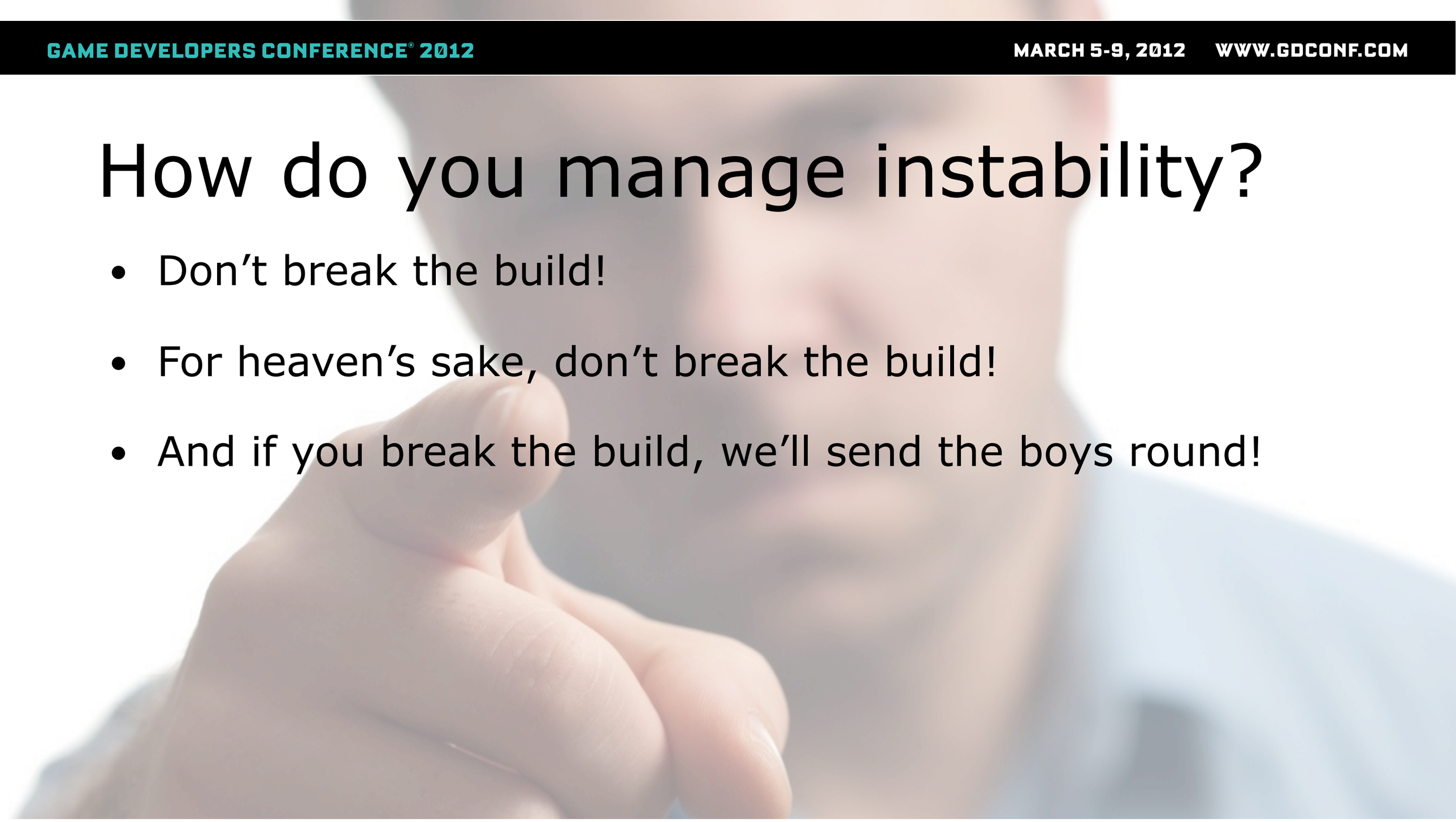
And the question that's being asked is: "how come the build is broken again?" And we revisit our programming practices and check-in procedures, to make sure that we don't break the build EVER again. Breakage is really costing us, so we put in place a code vetting system, where every code submission is peer reviewed. We put in place unit testing, smoke testing, branching, public flogging. All in an attempt to reduce build breakage.

But we are asking the wrong question. The question is not "how come the build is broken again?", because the answer to that is quite obvious. The build is broken because the game is still in development. Of course it is going to break. Software that is in development should be expected to break, a good deal of the time.

The real question is "when the build is broken, why do people have to sit on their hands?"

How do you manage instability?

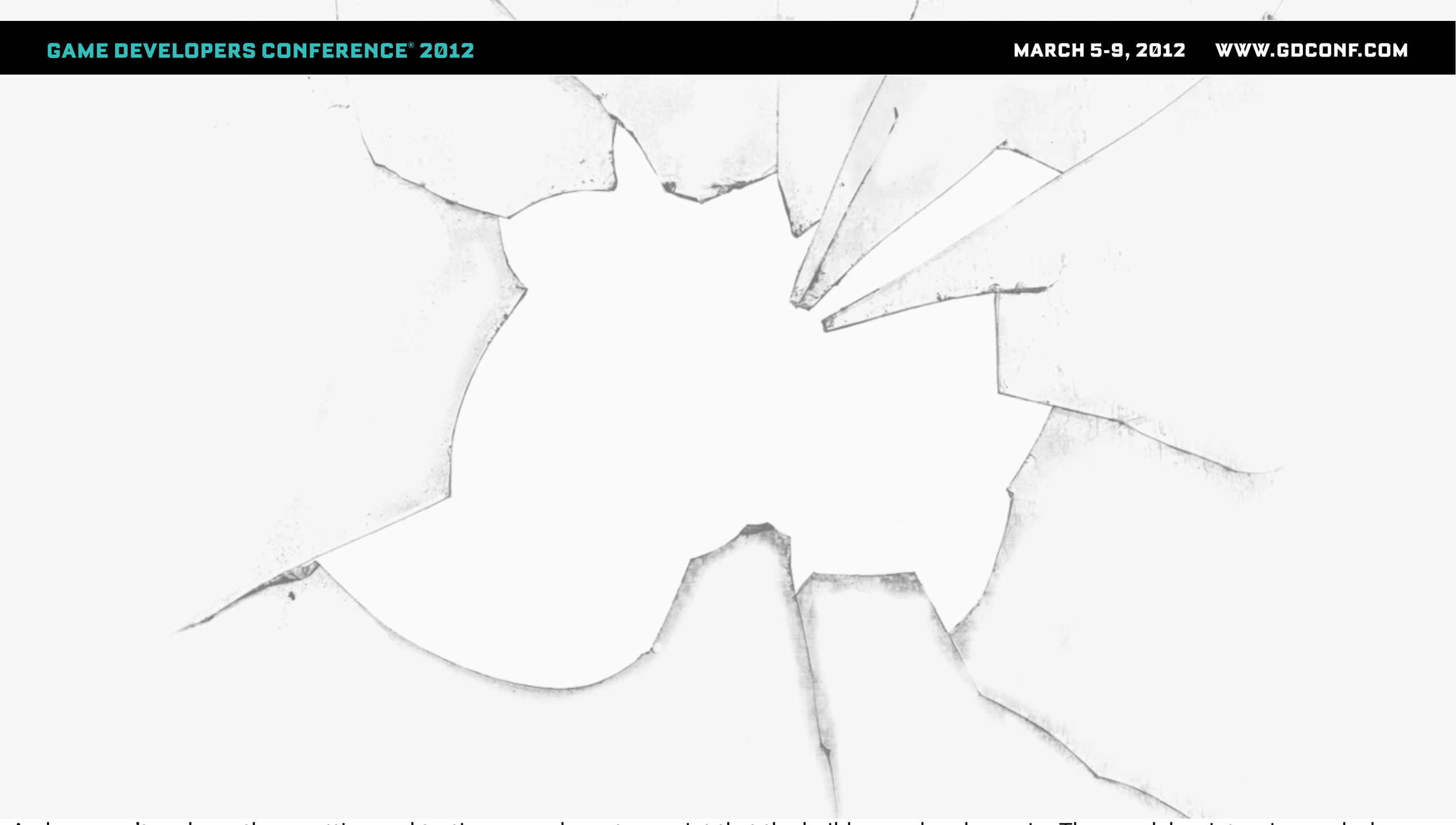
- Don’t break the build!
- For heaven’s sake, don’t break the build!
- And if you break the build, we’ll send the boys round!



And the question that's being asked is: "how come the build is broken again?" And we revisit our programming practices and check-in procedures, to make sure that we don't break the build EVER again. Breakage is really costing us, so we put in place a code vetting system, where every code submission is peer reviewed. We put in place unit testing, smoke testing, branching, public flogging. All in an attempt to reduce build breakage.

But we are asking the wrong question. The question is not "how come the build is broken again?", because the answer to that is quite obvious. The build is broken because the game is still in development. Of course it is going to break. Software that is in development should be expected to break, a good deal of the time.

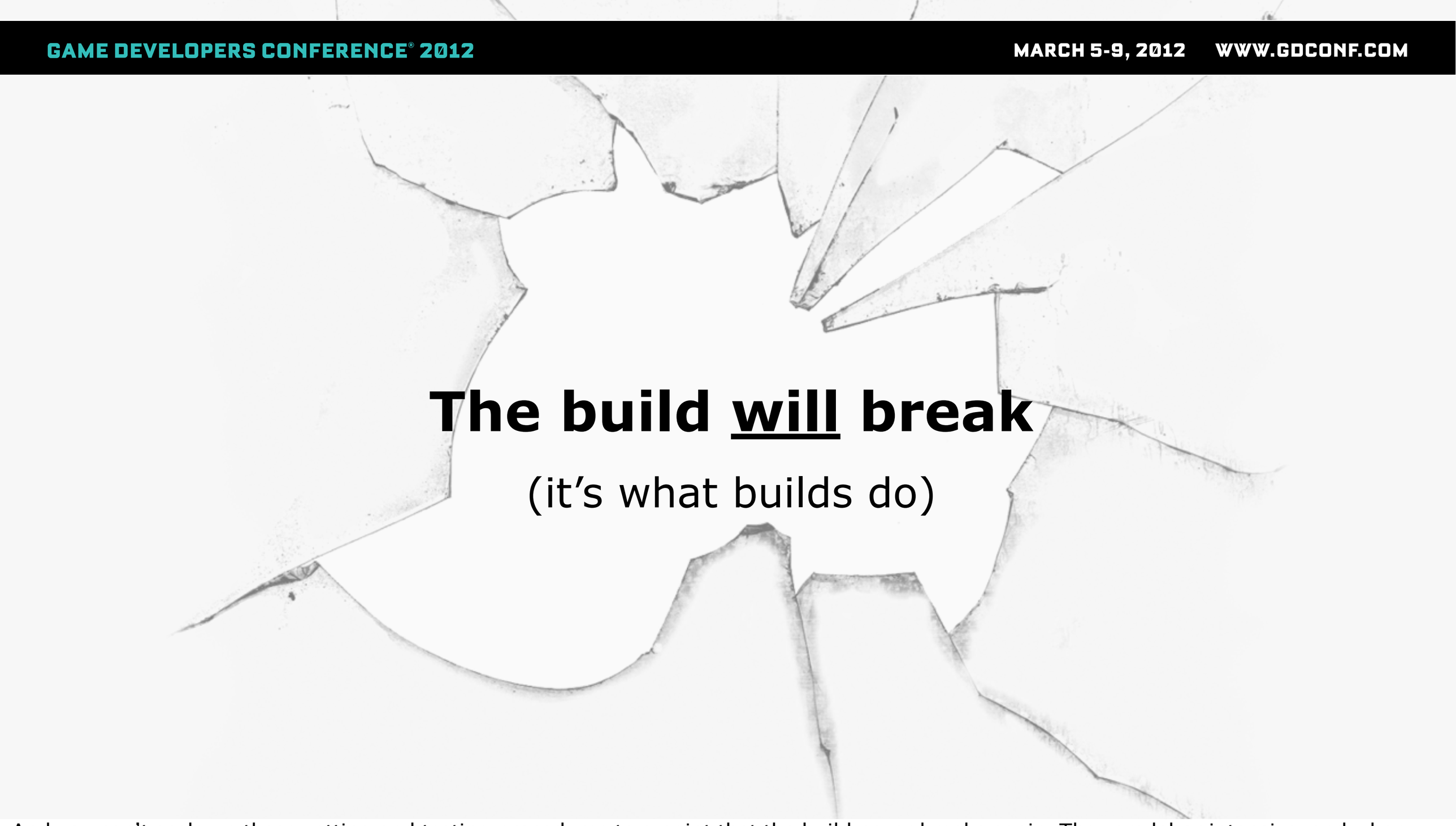
The real question is "when the build is broken, why do people have to sit on their hands?"



And you can't scale up these vetting and testing procedures to a point that the build never breaks again. They are labor intensive, and when taken to an extreme, they can slow down development and stifle creativity and experimentation. And the other problem is that they only address the FREQUENCY of breakages. It's as if we think that if we try hard enough, we will never break the build again. That of course is not the case. The build will still break.

I won't argue that we should do away with testing and branching and flogging. It is still important that we reduce the frequency of breakages.

But a plan that only addresses the FREQUENCY of breakage is incomplete.



And you can't scale up these vetting and testing procedures to a point that the build never breaks again. They are labor intensive, and when taken to an extreme, they can slow down development and stifle creativity and experimentation. And the other problem is that they only address the FREQUENCY of breakages. It's as if we think that if we try hard enough, we will never break the build again. That of course is not the case. The build will still break.

I won't argue that we should do away with testing and branching and flogging. It is still important that we reduce the frequency of breakages.

But a plan that only addresses the FREQUENCY of breakage is incomplete.

Downtime

You see, the problem is not really breakage. It is DOWNTIME. There are other factors at play here, not just the frequency. Overall downtime is a product of the frequency of breakage, and the average time it takes to fix each one.

Well actually that's wrong. This equation is very out of date.

[CLICK] This WAS the case in 1986, when I started out in the game industry. Back then, team size was typically ONE. "I" was the game designer, the artist, the sound guy, and also the programmer. "I was making a computer game." So if the build was broken, "I" had to fix it. And the time it took me to fix it was downtime. If it took me half a day to fix it, that was half a day of downtime.

Downtime

You see, the problem is not really breakage. It is DOWNTIME. There are other factors at play here, not just the frequency. Overall downtime is a product of the frequency of breakage, and the average time it takes to fix each one.

Well actually that's wrong. This equation is very out of date.

[CLICK] This WAS the case in 1986, when I started out in the game industry. Back then, team size was typically ONE. "I" was the game designer, the artist, the sound guy, and also the programmer. "I was making a computer game." So if the build was broken, "I" had to fix it. And the time it took me to fix it was downtime. If it took me half a day to fix it, that was half a day of downtime.

Downtime

You see, the problem is not really breakage. It is DOWNTIME. There are other factors at play here, not just the frequency. Overall downtime is a product of the frequency of breakage, and the average time it takes to fix each one.

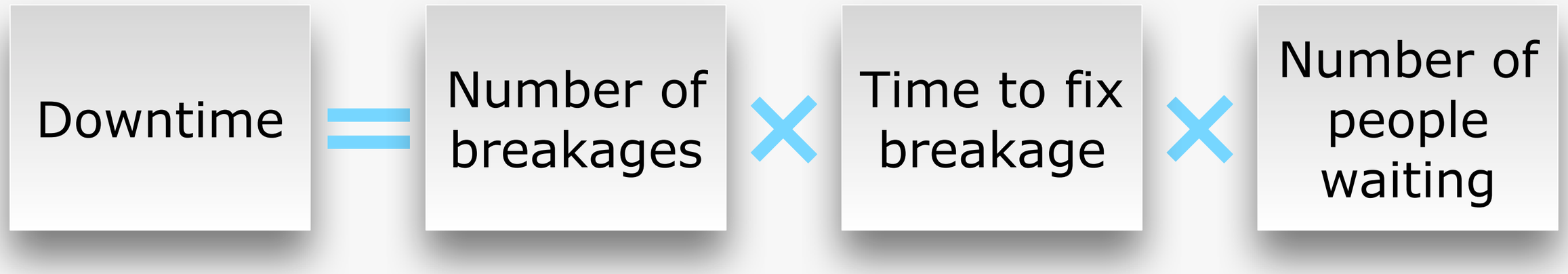
Well actually that's wrong. This equation is very out of date.

[CLICK] This WAS the case in 1986, when I started out in the game industry. Back then, team size was typically ONE. "I" was the game designer, the artist, the sound guy, and also the programmer. "I was making a computer game." So if the build was broken, "I" had to fix it. And the time it took me to fix it was downtime. If it took me half a day to fix it, that was half a day of downtime.

Downtime

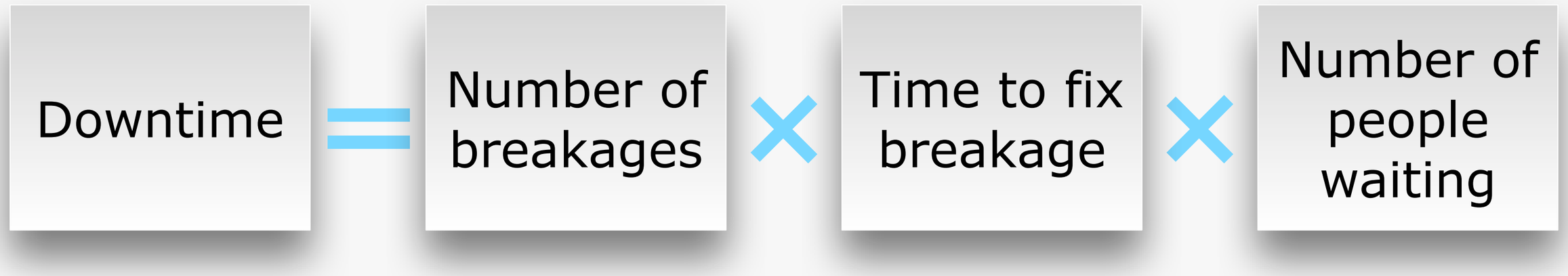
But that was a quarter of a century ago. Today we have team sizes of 50, 80, and more. When you have 50 people waiting for you to fix the build, and it takes you half a day, that is 25 man-days of downtime. That is more than a month.

Downtime



But that was a quarter of a century ago. Today we have team sizes of 50, 80, and more. When you have 50 people waiting for you to fix the build, and it takes you half a day, that is 25 man-days of downtime. That is more than a month.

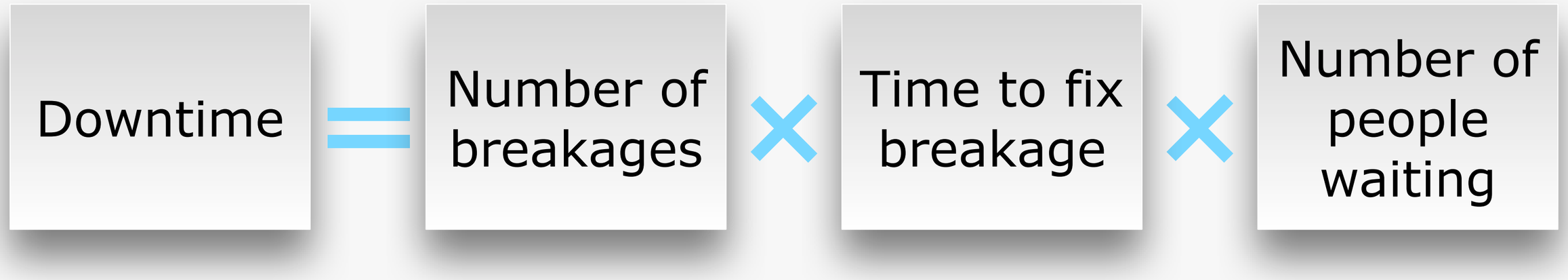
Downtime



So we clearly need to minimize DOWNTIME. If we can spend more time being productive, we can make the game more awesome, and still have time to spend with our families.

Downtime

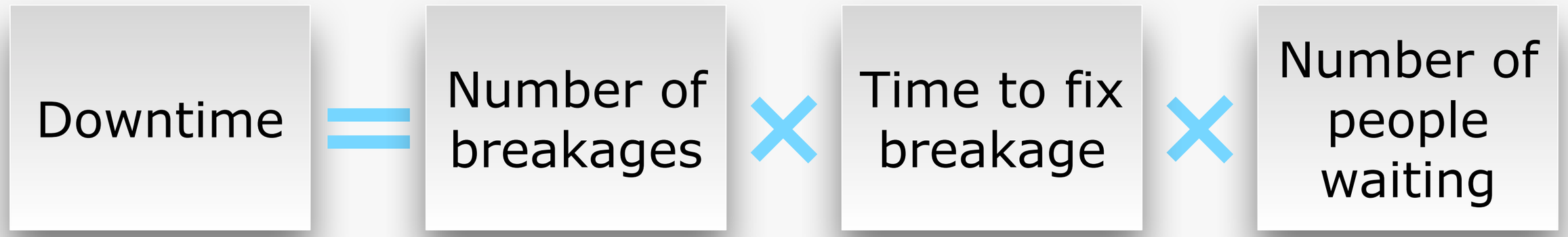
- Minimizing downtime is not the same as minimizing build breakage frequency



So we clearly need to minimize DOWNTIME. If we can spend more time being productive, we can make the game more awesome, and still have time to spend with our families.

Downtime

- Minimizing downtime is not the same as minimizing build breakage frequency
- Vetting, testing, branching only address occurrence of breakage

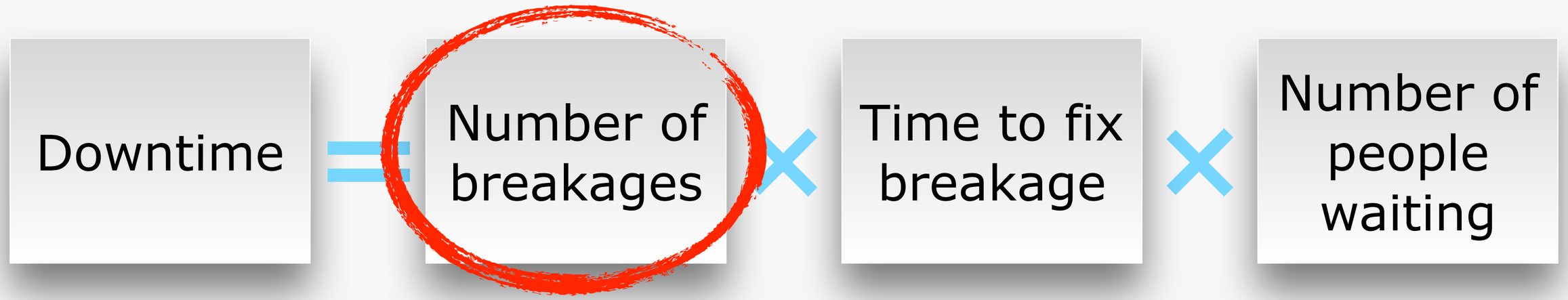


But it seems that most of our efforts into reducing downtime go to reducing the NUMBER of breakages.

This is what code vetting, testing, branching and flogging is all about.

Downtime

- Minimizing downtime is not the same as minimizing build breakage frequency
- Vetting, testing, branching only address occurrence of breakage

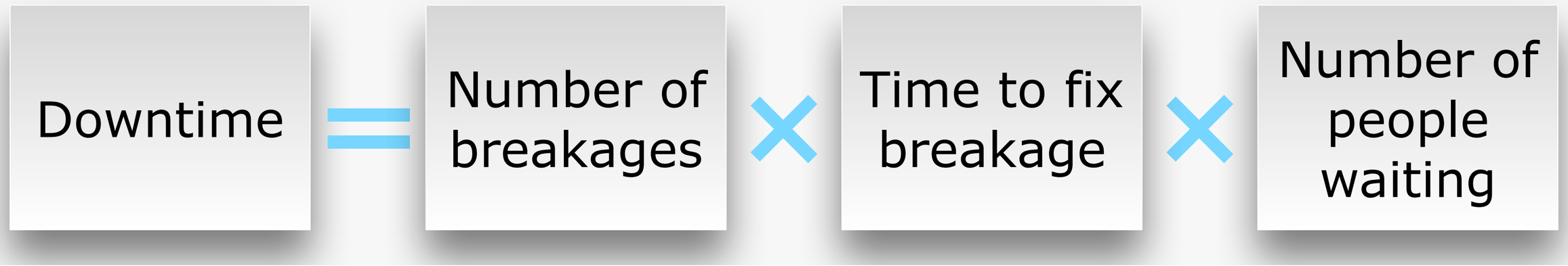


But it seems that most of our efforts into reducing downtime go to reducing the NUMBER of breakages.

This is what code vetting, testing, branching and flogging is all about.

Downtime

- Minimizing downtime is not the same as minimizing build breakage frequency
- Vetting, testing, branching only address occurrence of breakage
- Need to look at how breakage affects the team

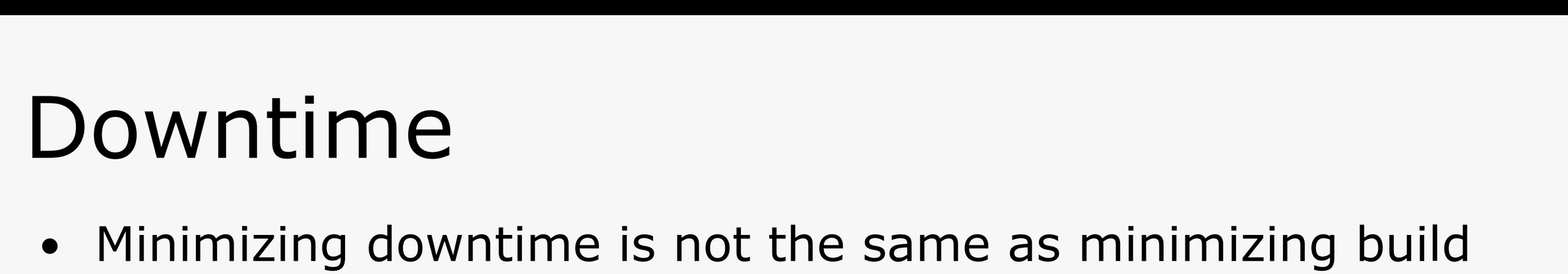


So today I want to shift the focus to measures you can take to reduce the number of people that breakage will affect. Note that in the slide I wrote “the number of people WAITING”, not “the number of people on the team”. If people can carry on working while you are fixing the build, there is no downtime.

[CLICK] And I will also talk a bit about reducing the time it takes to fix the build, by being better prepared.

Downtime

- Minimizing downtime is not the same as minimizing build breakage frequency
- Vetting, testing, branching only address occurrence of breakage
- Need to look at how breakage affects the team

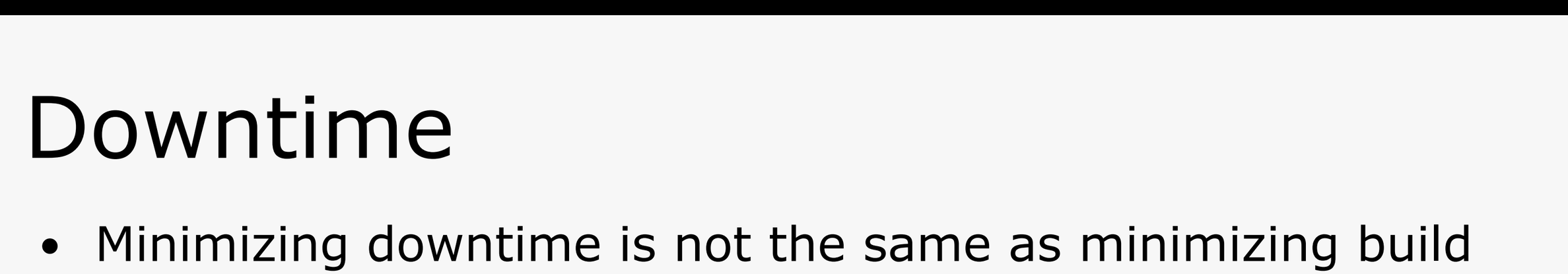


So today I want to shift the focus to measures you can take to reduce the number of people that breakage will affect. Note that in the slide I wrote “the number of people WAITING”, not “the number of people on the team”. If people can carry on working while you are fixing the build, there is no downtime.

[CLICK] And I will also talk a bit about reducing the time it takes to fix the build, by being better prepared.

Downtime

- Minimizing downtime is not the same as minimizing build breakage frequency
- Vetting, testing, branching only address occurrence of breakage
- Need to look at how breakage affects the team



So today I want to shift the focus to measures you can take to reduce the number of people that breakage will affect. Note that in the slide I wrote “the number of people WAITING”, not “the number of people on the team”. If people can carry on working while you are fixing the build, there is no downtime.

[CLICK] And I will also talk a bit about reducing the time it takes to fix the build, by being better prepared.

Three downtime reducing measures

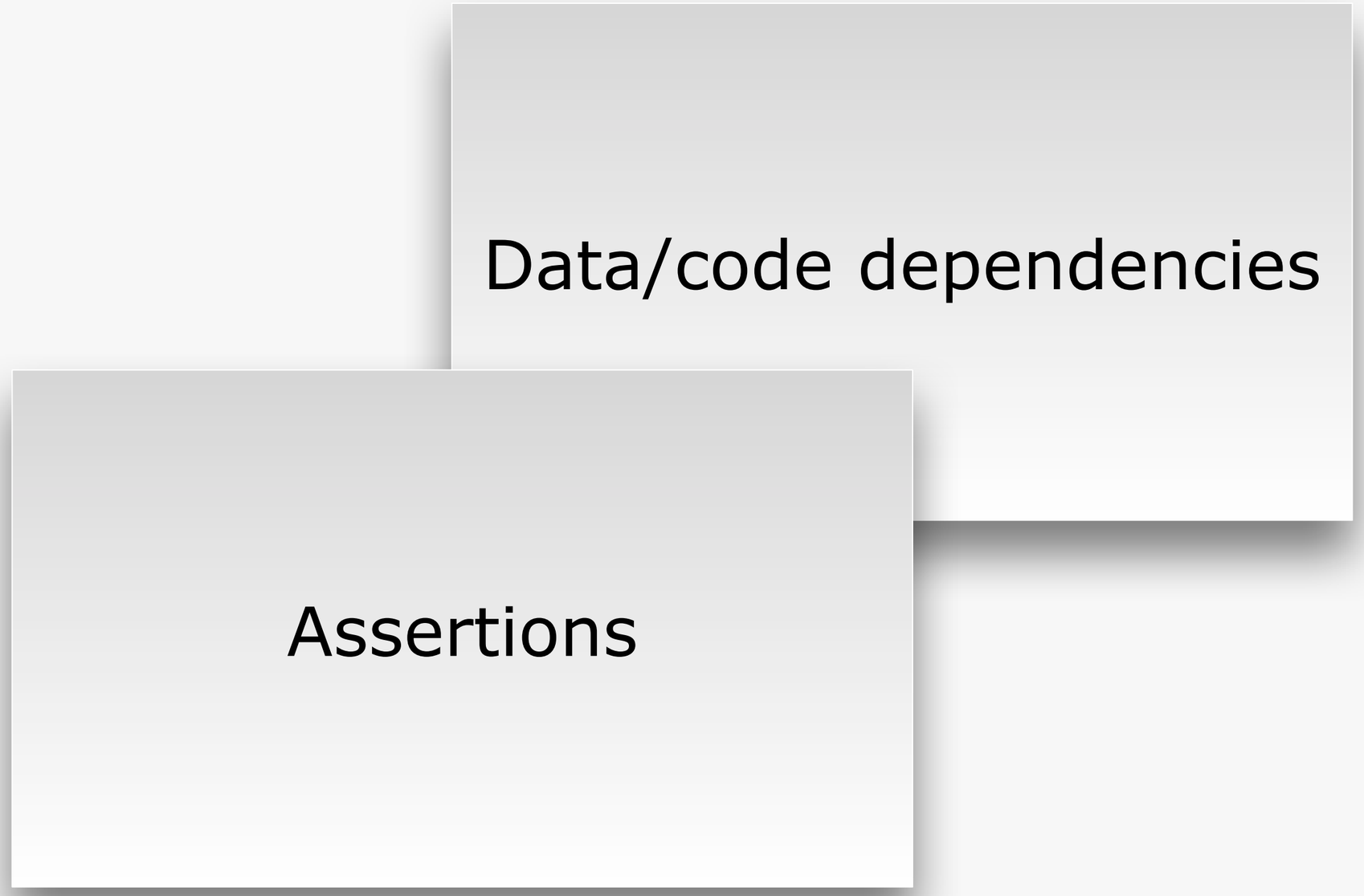
In this session I will highlight three downtime reducing measures. And these are just examples. I hope that by the end of this session, you will have picked up some ideas, but most of all, that you are inspired to look at your own production pipeline, and make it fail in a better way.

[CLICK] Next up, I will talk about data/code dependencies. I found that the choices we make HERE have a profound effect on productivity.
[CLICK] Then I will talk about assertions, how to make them less intrusive and yet more effective.
[CLICK] And finally I will discuss how, at Insomniac, we have successfully prevented loss of work in our tools due to crashes, by using a client/server model, and got a ton of goodies for free.

Data/code dependencies

In this session I will highlight three downtime reducing measures. And these are just examples. I hope that by the end of this session, you will have picked up some ideas, but most of all, that you are inspired to look at your own production pipeline, and make it fail in a better way.

[CLICK] Next up, I will talk about data/code dependencies. I found that the choices we make HERE have a profound effect on productivity.
[CLICK] Then I will talk about assertions, how to make them less intrusive and yet more effective.
[CLICK] And finally I will discuss how, at Insomniac, we have successfully prevented loss of work in our tools due to crashes, by using a client/server model, and got a ton of goodies for free.



In this session I will highlight three downtime reducing measures. And these are just examples. I hope that by the end of this session, you will have picked up some ideas, but most of all, that you are inspired to look at your own production pipeline, and make it fail in a better way.

[CLICK] Next up, I will talk about data/code dependencies. I found that the choices we make HERE have a profound effect on productivity.
[CLICK] Then I will talk about assertions, how to make them less intrusive and yet more effective.
[CLICK] And finally I will discuss how, at Insomniac, we have successfully prevented loss of work in our tools due to crashes, by using a client/server model, and got a ton of goodies for free.

Data/code dependencies

Assertions

Client/server tools architecture

In this session I will highlight three downtime reducing measures. And these are just examples. I hope that by the end of this session, you will have picked up some ideas, but most of all, that you are inspired to look at your own production pipeline, and make it fail in a better way.

[CLICK] Next up, I will talk about data/code dependencies. I found that the choices we make HERE have a profound effect on productivity.
[CLICK] Then I will talk about assertions, how to make them less intrusive and yet more effective.
[CLICK] And finally I will discuss how, at Insomniac, we have successfully prevented loss of work in our tools due to crashes, by using a client/server model, and got a ton of goodies for free.

Data/code dependencies

Mercenaries, The Saboteur, and Resistance 3 all suffered from downtime caused by data/code dependency issues.

Data build times

Lengthy data rebuilds a never fun. But they can become a real issue when the build is broken.
If the latest version of the game executable requires a new data version, that is what I would call a REGULAR data rebuild, and you can plan it into your day. You set it going overnight, or over lunch.
[PAUSE] When the build is broken, the first line of defense against downtime, is to offer an archive of previous builds. For artists and designers, yesterday's build is a good fall-back option. Should take only a few minutes, no downtime.
[CLICK] But if yesterday's executable requires yesterday's data format, you are faced with a forced data rebuild.
And NOW switching back to yesterday's executable takes two hours. And you can't plan it into your day. So that's real downtime. And later, when the build is fixed, another two hours to switch back to today's data format.
[CLICK] And this is what happened at The Saboteur. There was a strong dependency between the code version and the data version, so when the build was broken, people would rather wait for a fix than face the additional two data rebuilds they would need to switch back to yesterday's executable.
Mercenaries and the Saboteur had very different approaches to data formats and data building, and I will compare the two. Out of these two, there is no overall winner. But stay tuned, I may have a solution.

Data build times

- "Regular" data builds can be made to fit in your work schedule

Lengthy data rebuilds a never fun. But they can become a real issue when the build is broken.
If the latest version of the game executable requires a new data version, that is what I would call a REGULAR data rebuild, and you can plan it into your day. You set it going overnight, or over lunch.
[PAUSE] When the build is broken, the first line of defense against downtime, is to offer an archive of previous builds. For artists and designers, yesterday's build is a good fall-back option. Should take only a few minutes, no downtime.
[CLICK] But if yesterday's executable requires yesterday's data format, you are faced with a forced data rebuild.
And NOW switching back to yesterday's executable takes two hours. And you can't plan it into your day. So that's real downtime. And later, when the build is fixed, another two hours to switch back to today's data format.
[CLICK] And this is what happened at The Saboteur. There was a strong dependency between the code version and the data version, so when the build was broken, people would rather wait for a fix than face the additional two data rebuilds they would need to switch back to yesterday's executable.
Mercenaries and the Saboteur had very different approaches to data formats and data building, and I will compare the two. Out of these two, there is no overall winner. But stay tuned, I may have a solution.

Data build times

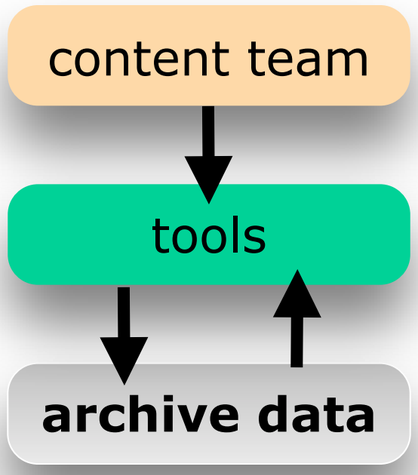
- "Regular" data builds can be made to fit in your work schedule
- "Forced" data builds are intrusive

Lengthy data rebuilds a never fun. But they can become a real issue when the build is broken.
If the latest version of the game executable requires a new data version, that is what I would call a REGULAR data rebuild, and you can plan it into your day. You set it going overnight, or over lunch.
[PAUSE] When the build is broken, the first line of defense against downtime, is to offer an archive of previous builds. For artists and designers, yesterday's build is a good fall-back option. Should take only a few minutes, no downtime.
[CLICK] But if yesterday's executable requires yesterday's data format, you are faced with a forced data rebuild.
And NOW switching back to yesterday's executable takes two hours. And you can't plan it into your day. So that's real downtime. And later, when the build is fixed, another two hours to switch back to today's data format.
[CLICK] And this is what happened at The Saboteur. There was a strong dependency between the code version and the data version, so when the build was broken, people would rather wait for a fix than face the additional two data rebuilds they would need to switch back to yesterday's executable.
Mercenaries and the Saboteur had very different approaches to data formats and data building, and I will compare the two. Out of these two, there is no overall winner. But stay tuned, I may have a solution.

Data build times

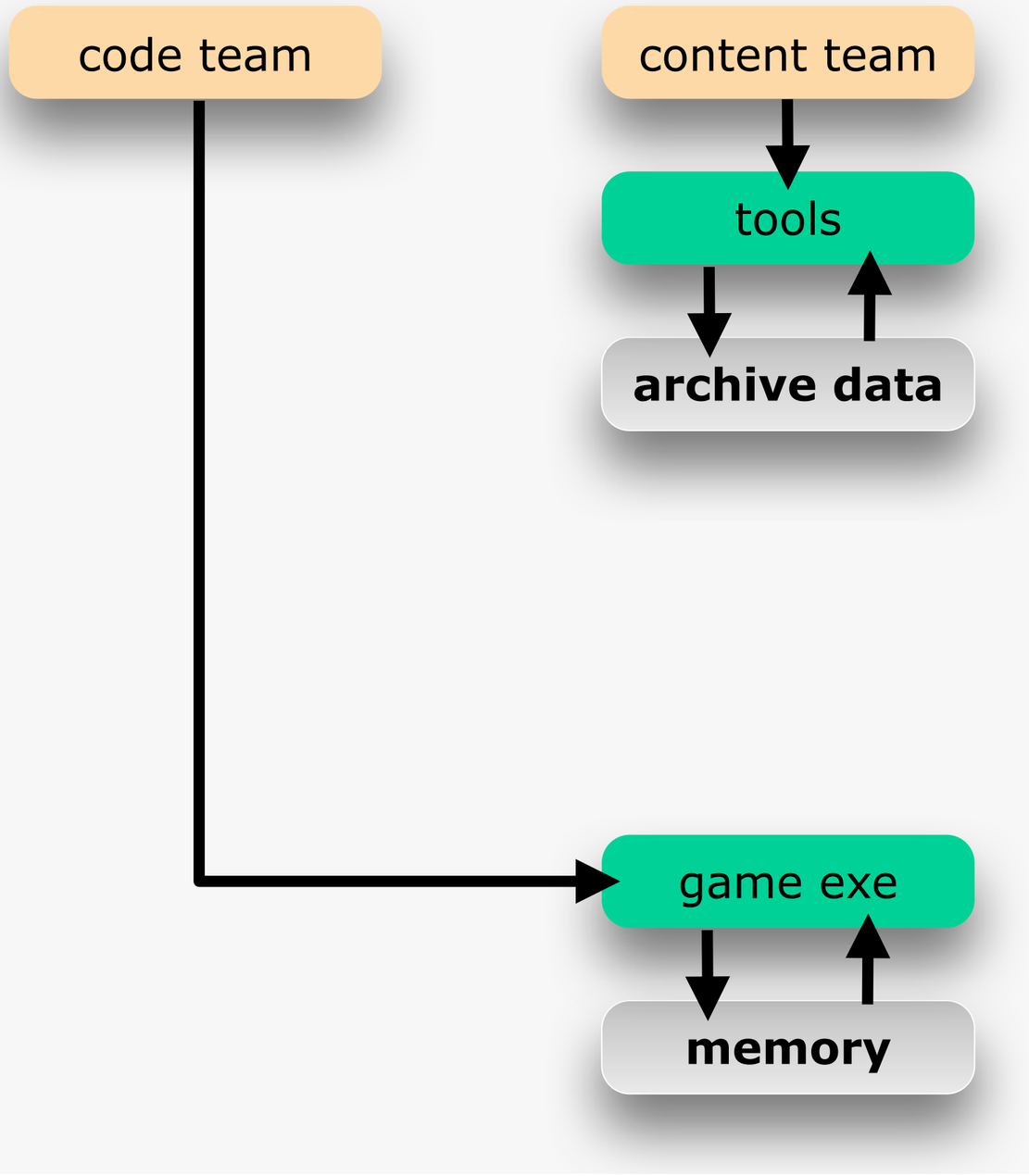
- "Regular" data builds can be made to fit in your work schedule
- "Forced" data builds are intrusive
- Forced data rebuilds are caused by data/code dependencies

Lengthy data rebuilds a never fun. But they can become a real issue when the build is broken.
If the latest version of the game executable requires a new data version, that is what I would call a REGULAR data rebuild, and you can plan it into your day. You set it going overnight, or over lunch.
[PAUSE] When the build is broken, the first line of defense against downtime, is to offer an archive of previous builds. For artists and designers, yesterday's build is a good fall-back option. Should take only a few minutes, no downtime.
[CLICK] But if yesterday's executable requires yesterday's data format, you are faced with a forced data rebuild.
And NOW switching back to yesterday's executable takes two hours. And you can't plan it into your day. So that's real downtime. And later, when the build is fixed, another two hours to switch back to today's data format.
[CLICK] And this is what happened at The Saboteur. There was a strong dependency between the code version and the data version, so when the build was broken, people would rather wait for a fix than face the additional two data rebuilds they would need to switch back to yesterday's executable.
Mercenaries and the Saboteur had very different approaches to data formats and data building, and I will compare the two. Out of these two, there is no overall winner. But stay tuned, I may have a solution.



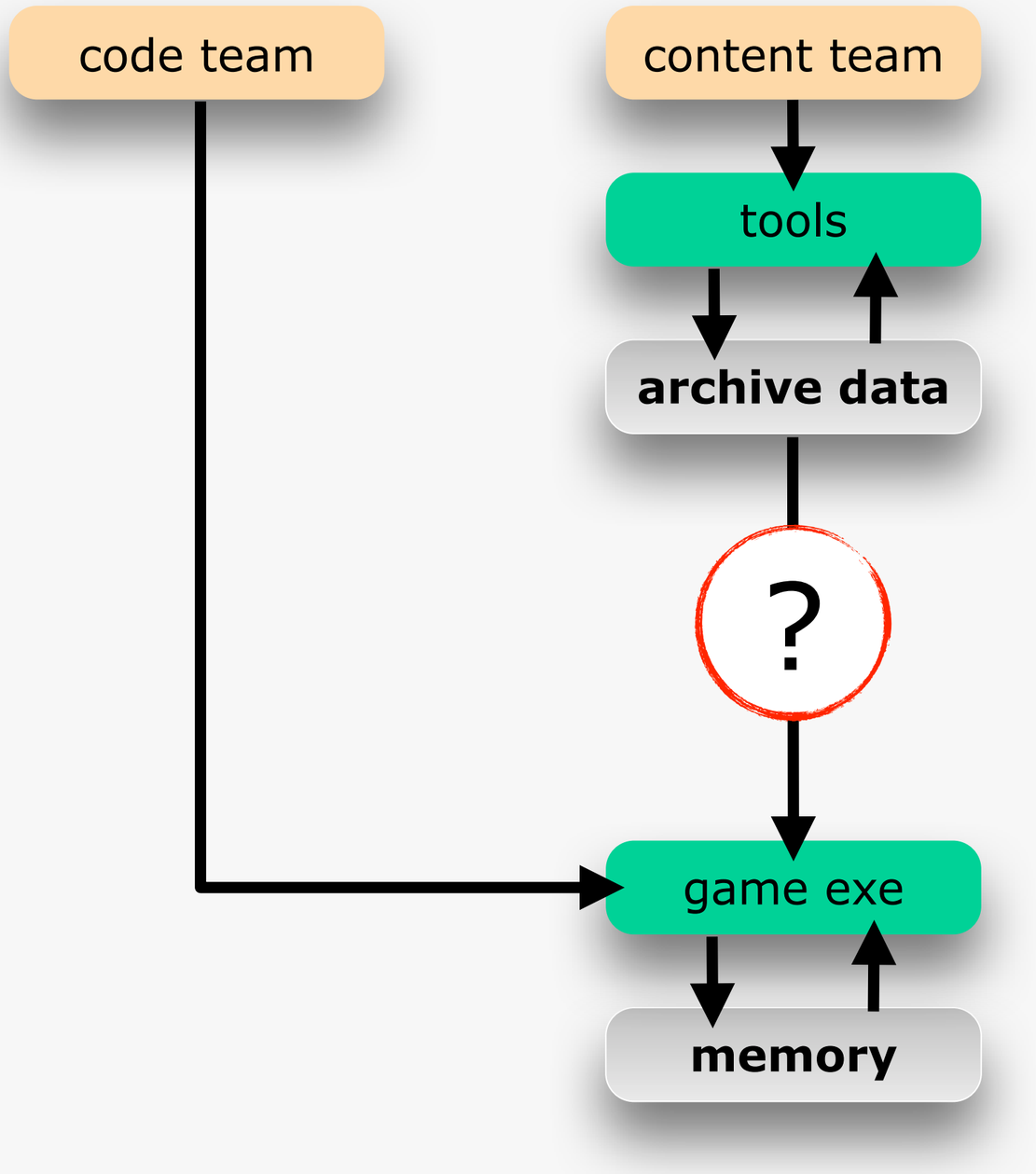
So first let me show you the production pipeline. Mercenaries and the Saboteur looked very similar in this respect. I found the same thing again when I joined Insomniac. I believe this is a common pattern.

The content team, that's the artists and designers, work with third party and in-house tools. These tools read and write their own native formats. This is what I call "archive data". Some call it "source data". This is a collection of formats that we generally have no control over, and that can be written as well as read by the tools.



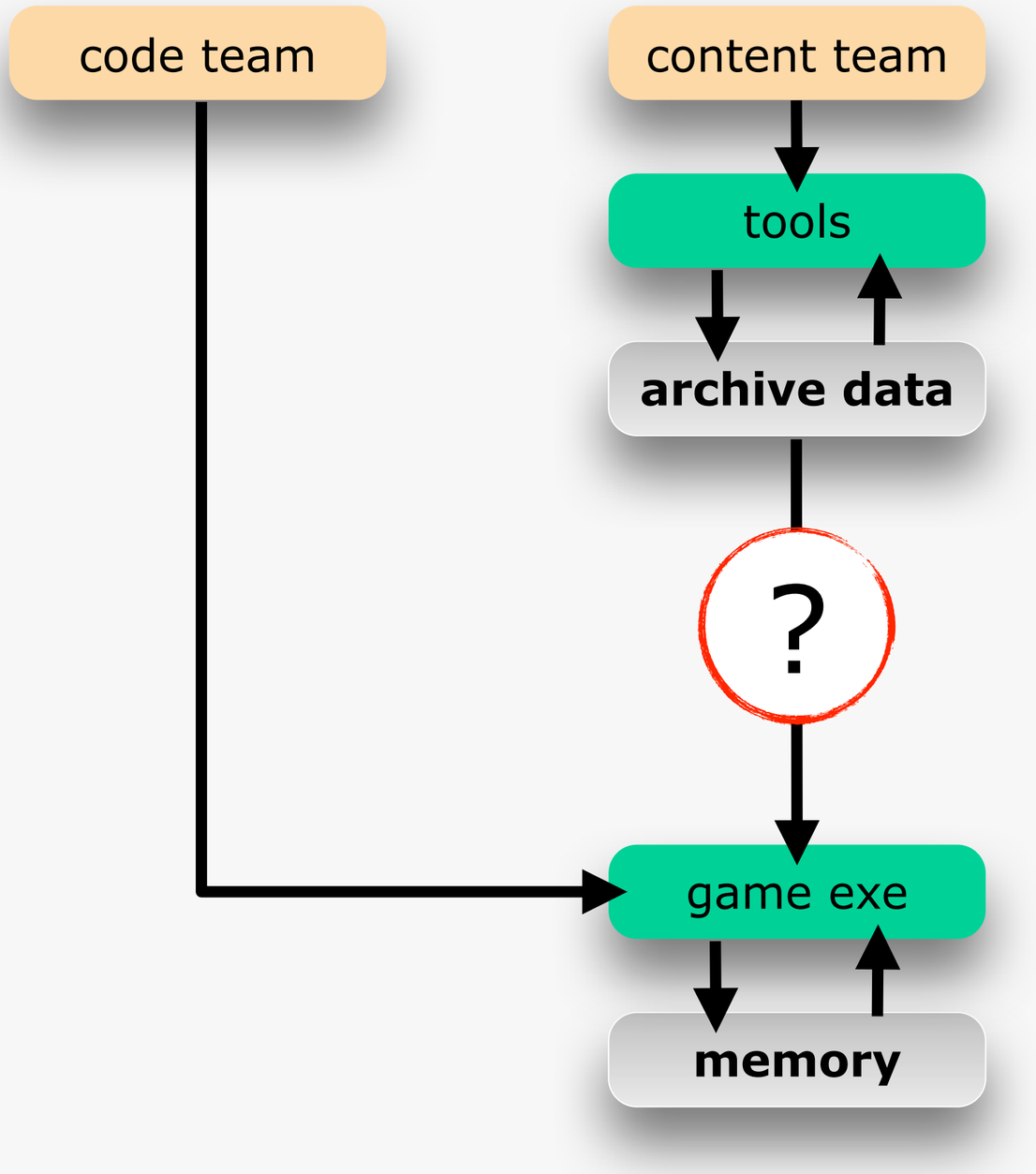
Eventually, data from these files has to end up in the game's memory. At runtime, we don't want to be parsing a hodge-podge of archive data formats, if we even could.

[CLICK] So pretty much all games transform this archive data into something a little more runtime digestible. I call that "engine data". This is optimized for loading at runtime, or streaming.

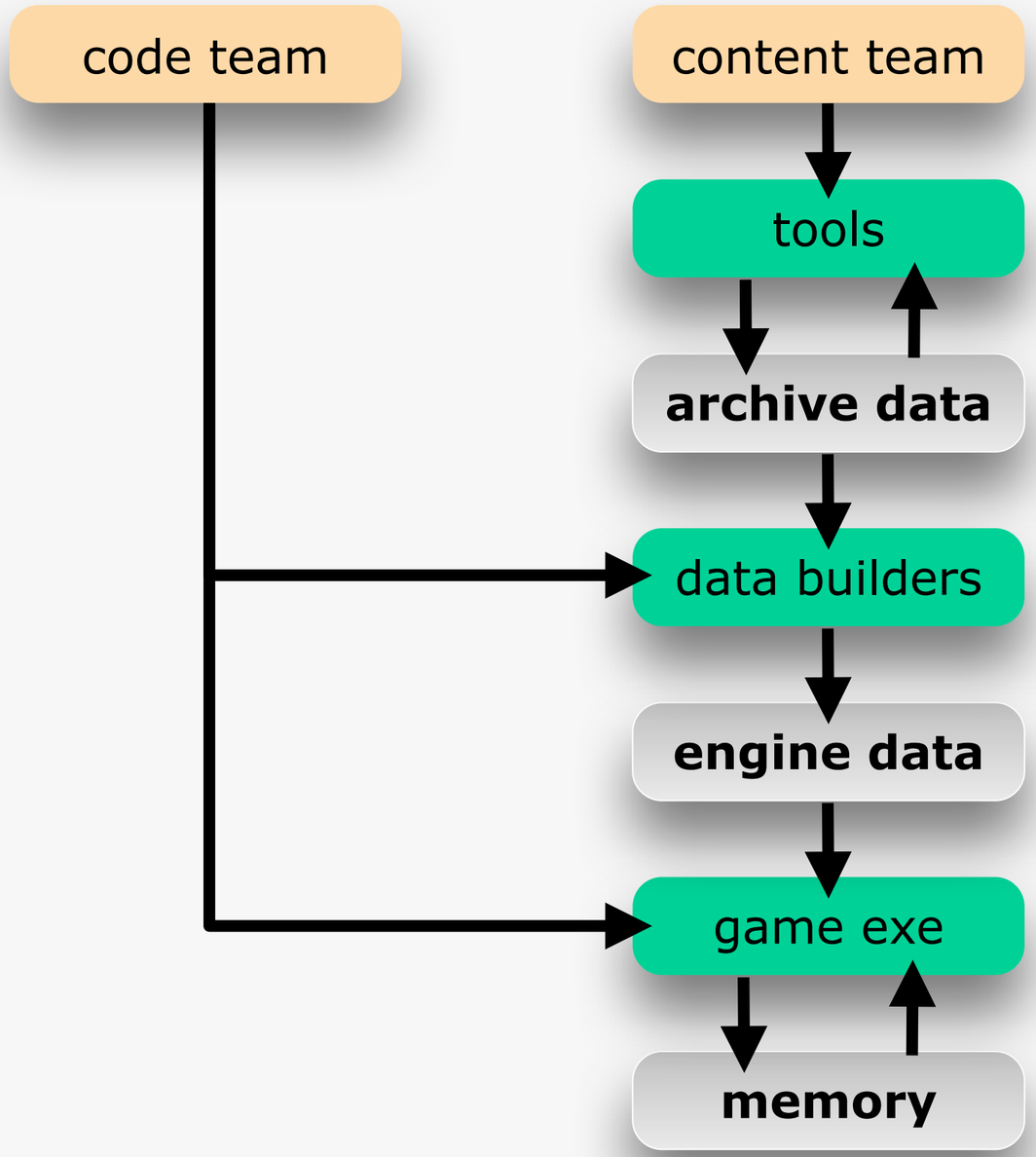


Eventually, data from these files has to end up in the game's memory. At runtime, we don't want to be parsing a hodge-podge of archive data formats, if we even could.

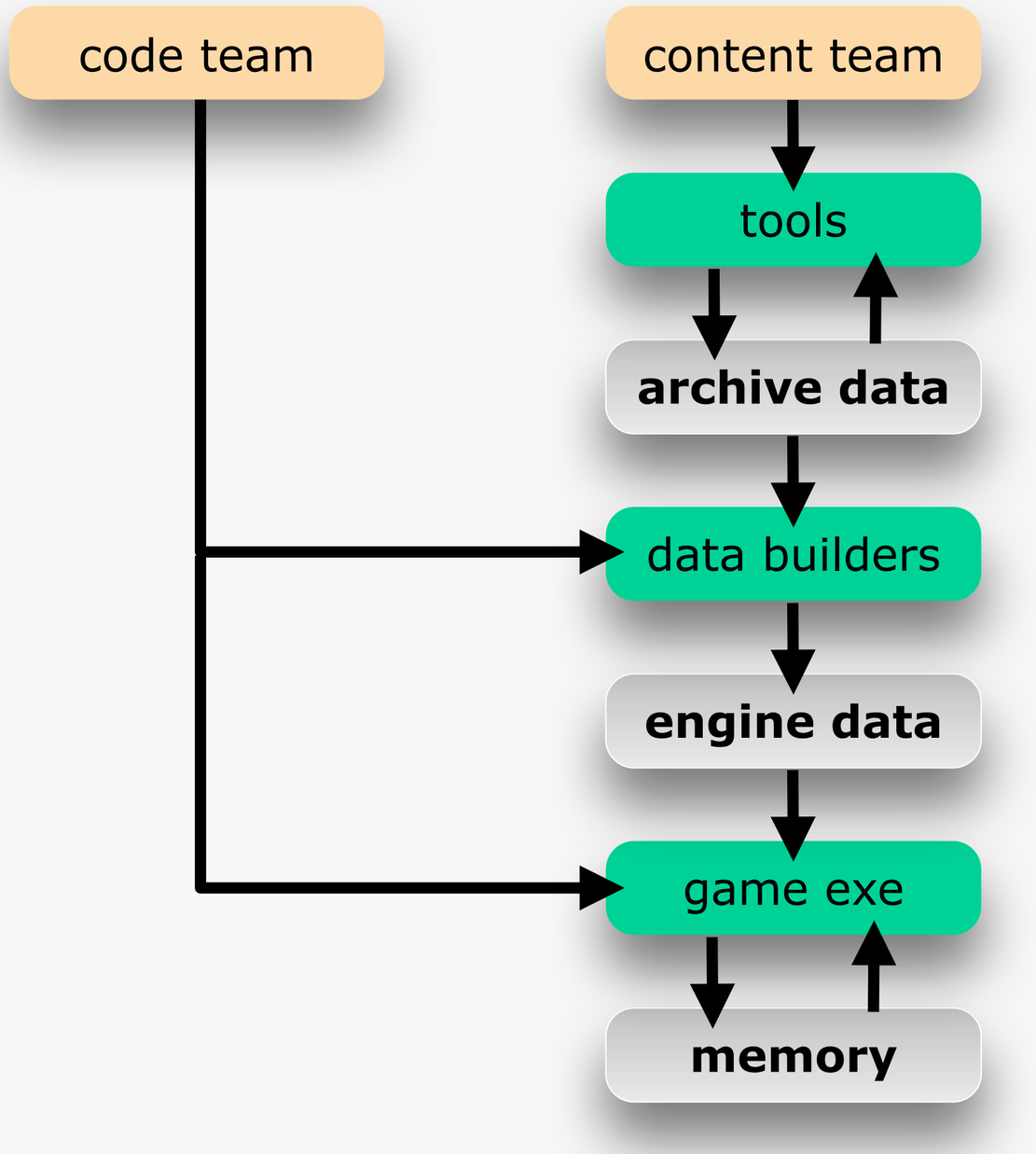
[CLICK] So pretty much all games transform this archive data into something a little more runtime digestible. I call that "engine data". This is optimized for loading at runtime, or streaming.



And this is where the data builders come in.

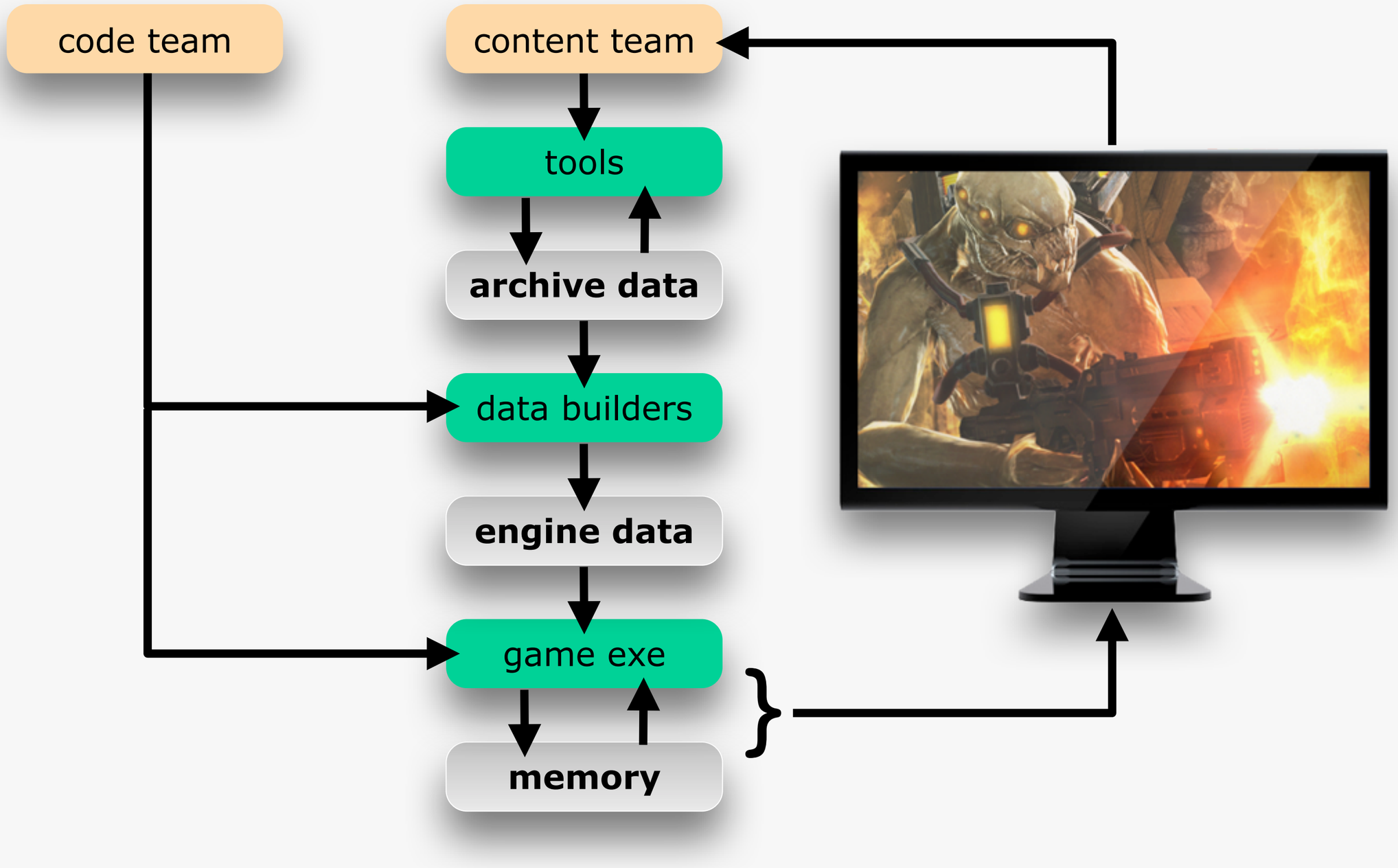


And this is where the data builders come in.



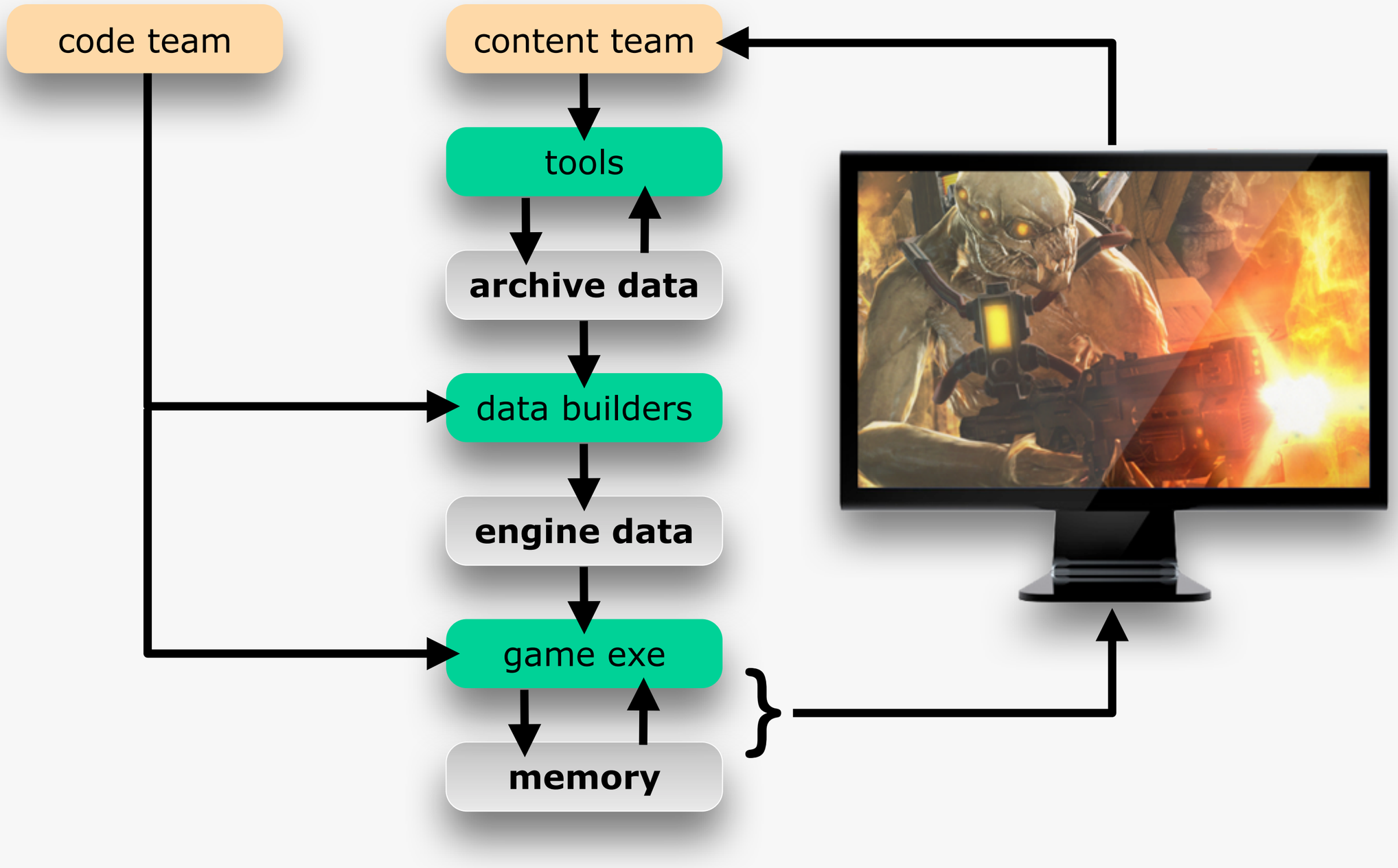
Most of the pipelines I have seen over the past decade or so have looked very much like this. But where they differed the most was the choice of the engine data format.

[CLICK] Both Mercenaries and The Saboteur were console games that made great demands on disk data transfer. Both were streaming open world games. When you are loading a level, as opposed to streaming, disk efficiency is just a matter of good manners: don't keep your players waiting. But in a streaming open world game, file performance affects the quality of the environment. As the player is speeding through the world in his convertible, models and texture, chunks of terrain and the like have to come in from disk. If the disk i/o doesn't keep up, you end up with artifacts, low res artwork in close-up, or worse, objects will be missing.



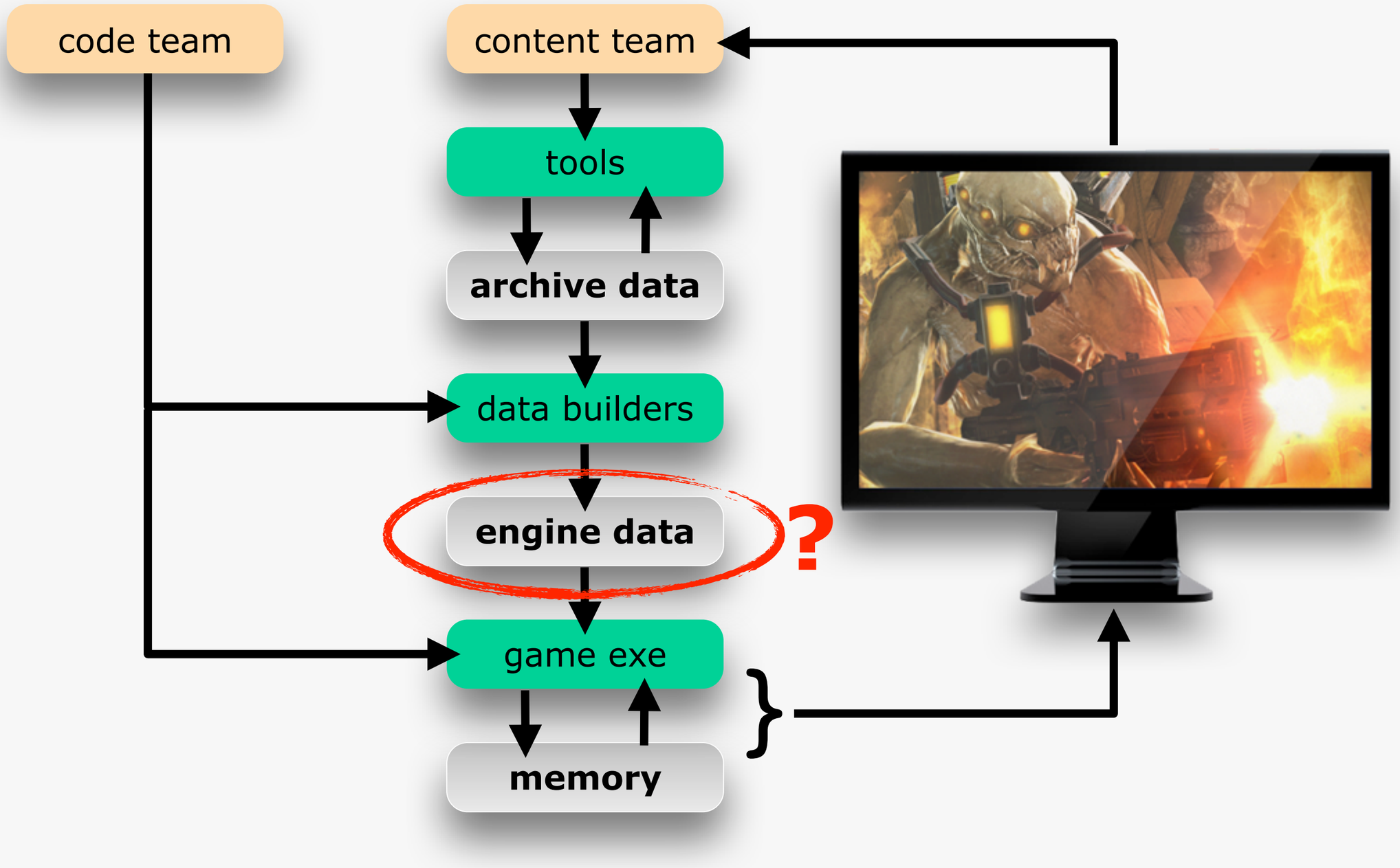
Most of the pipelines I have seen over the past decade or so have looked very much like this. But where they differed the most was the choice of the engine data format.

[CLICK] Both Mercenaries and The Saboteur were console games that made great demands on disk data transfer. Both were streaming open world games. When you are loading a level, as opposed to streaming, disk efficiency is just a matter of good manners: don't keep your players waiting. But in a streaming open world game, file performance affects the quality of the environment. As the player is speeding through the world in his convertible, models and texture, chunks of terrain and the like have to come in from disk. If the disk i/o doesn't keep up, you end up with artifacts, low res artwork in close-up, or worse, objects will be missing.



Most of the pipelines I have seen over the past decade or so have looked very much like this. But where they differed the most was the choice of the engine data format.

[CLICK] Both Mercenaries and The Saboteur were console games that made great demands on disk data transfer. Both were streaming open world games. When you are loading a level, as opposed to streaming, disk efficiency is just a matter of good manners: don't keep your players waiting. But in a streaming open world game, file performance affects the quality of the environment. As the player is speeding through the world in his convertible, models and texture, chunks of terrain and the like have to come in from disk. If the disk i/o doesn't keep up, you end up with artifacts, low res artwork in close-up, or worse, objects will be missing.



Most of the pipelines I have seen over the past decade or so have looked very much like this. But where they differed the most was the choice of the engine data format.

[CLICK] Both Mercenaries and The Saboteur were console games that made great demands on disk data transfer. Both were streaming open world games. When you are loading a level, as opposed to streaming, disk efficiency is just a matter of good manners: don't keep your players waiting. But in a streaming open world game, file performance affects the quality of the environment. As the player is speeding through the world in his convertible, models and texture, chunks of terrain and the like have to come in from disk. If the disk i/o doesn't keep up, you end up with artifacts, low res artwork in close-up, or worse, objects will be missing.

The Saboteur placed great emphasis on streaming efficiency. And they achieved that. They employed a technique that I will call load-n-go. Efficient for streaming, but strong dependencies.

[CLICK] Mercenaries on the other hand employed what I will call a READ-N-BUILD approach. This reduces data/code dependencies, and leads to fewer forced data rebuilds, but performance kind of sucks.

[CLICK] So I tried to find an approach that combines the efficiency of LOAD-N-GO without the dependency issues. I couldn't find it so I made one up. I call it STRUCTURED BINARY. This is, right now, an experimental technology, but it's looking promising.

Game data format categories

The Saboteur placed great emphasis on streaming efficiency. And they achieved that. They employed a technique that I will call load-n-go. Efficient for streaming, but strong dependencies.

[CLICK] Mercenaries on the other hand employed what I will call a READ-N-BUILD approach. This reduces data/code dependencies, and leads to fewer forced data rebuilds, but performance kind of sucks.

[CLICK] So I tried to find an approach that combines the efficiency of LOAD-N-GO without the dependency issues. I couldn't find it so I made one up. I call it STRUCTURED BINARY. This is, right now, an experimental technology, but it's looking promising.

Game data format categories

- “Load-n-Go”

The Saboteur placed great emphasis on streaming efficiency. And they achieved that. They employed a technique that I will call load-n-go. Efficient for streaming, but strong dependencies.

[CLICK] Mercenaries on the other hand employed what I will call a READ-N-BUILD approach. This reduces data/code dependencies, and leads to fewer forced data rebuilds, but performance kind of sucks.

[CLICK] So I tried to find an approach that combines the efficiency of LOAD-N-GO without the dependency issues. I couldn’t find it so I made one up. I call it STRUCTURED BINARY. This is, right now, an experimental technology, but it’s looking promising.

Game data format categories

- "Load-n-Go"
- "Read-n-Build"

The Saboteur placed great emphasis on streaming efficiency. And they achieved that. They employed a technique that I will call load-n-go. Efficient for streaming, but strong dependencies.

[CLICK] Mercenaries on the other hand employed what I will call a READ-N-BUILD approach. This reduces data/code dependencies, and leads to fewer forced data rebuilds, but performance kind of sucks.

[CLICK] So I tried to find an approach that combines the efficiency of LOAD-N-GO without the dependency issues. I couldn't find it so I made one up. I call it STRUCTURED BINARY. This is, right now, an experimental technology, but it's looking promising.

Game data format categories

- "Load-n-Go"
- "Read-n-Build"
- "Structured Binary"

The Saboteur placed great emphasis on streaming efficiency. And they achieved that. They employed a technique that I will call load-n-go. Efficient for streaming, but strong dependencies.

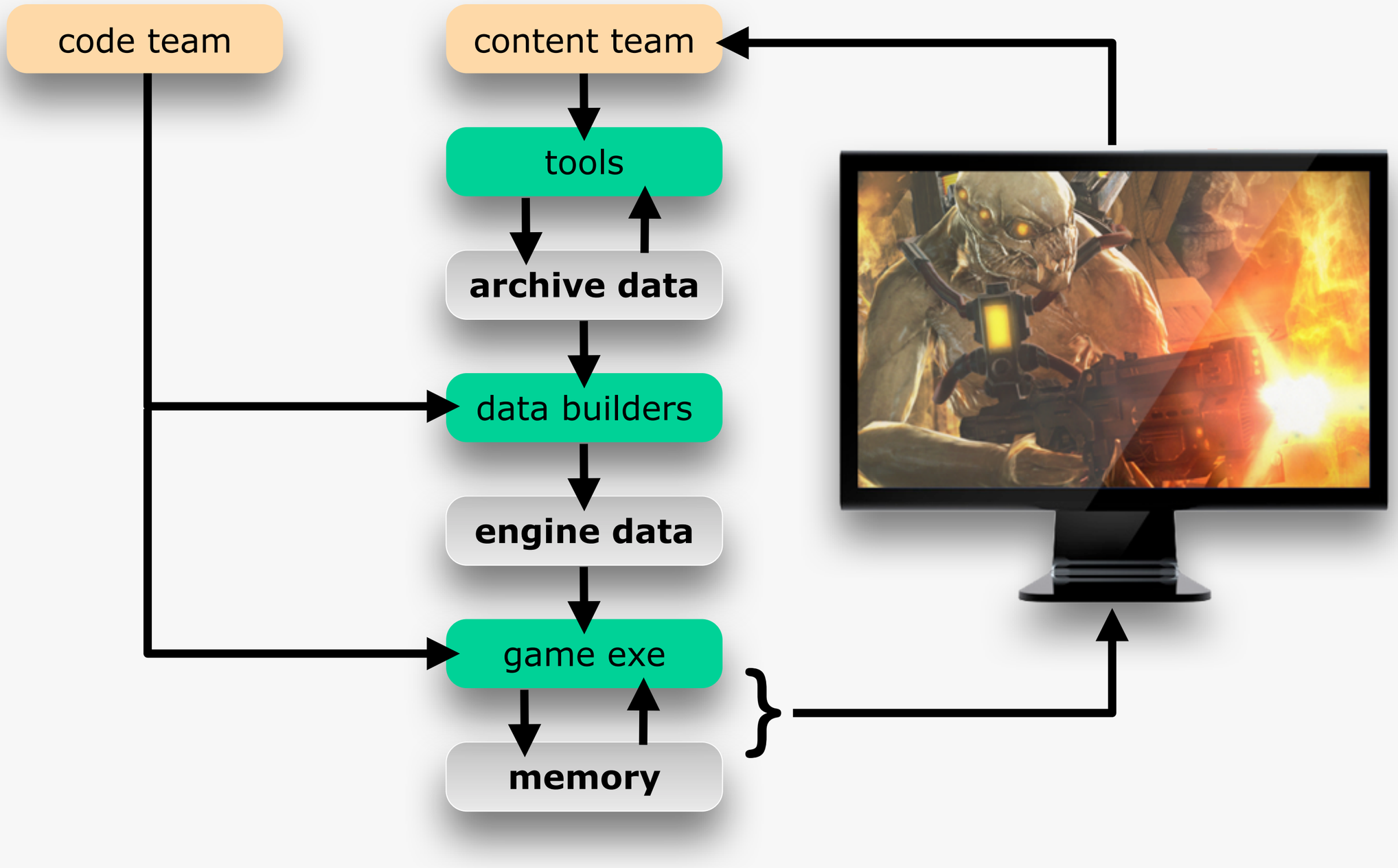
[CLICK] Mercenaries on the other hand employed what I will call a READ-N-BUILD approach. This reduces data/code dependencies, and leads to fewer forced data rebuilds, but performance kind of sucks.

[CLICK] So I tried to find an approach that combines the efficiency of LOAD-N-GO without the dependency issues. I couldn't find it so I made one up. I call it STRUCTURED BINARY. This is, right now, an experimental technology, but it's looking promising.

Game data format categories

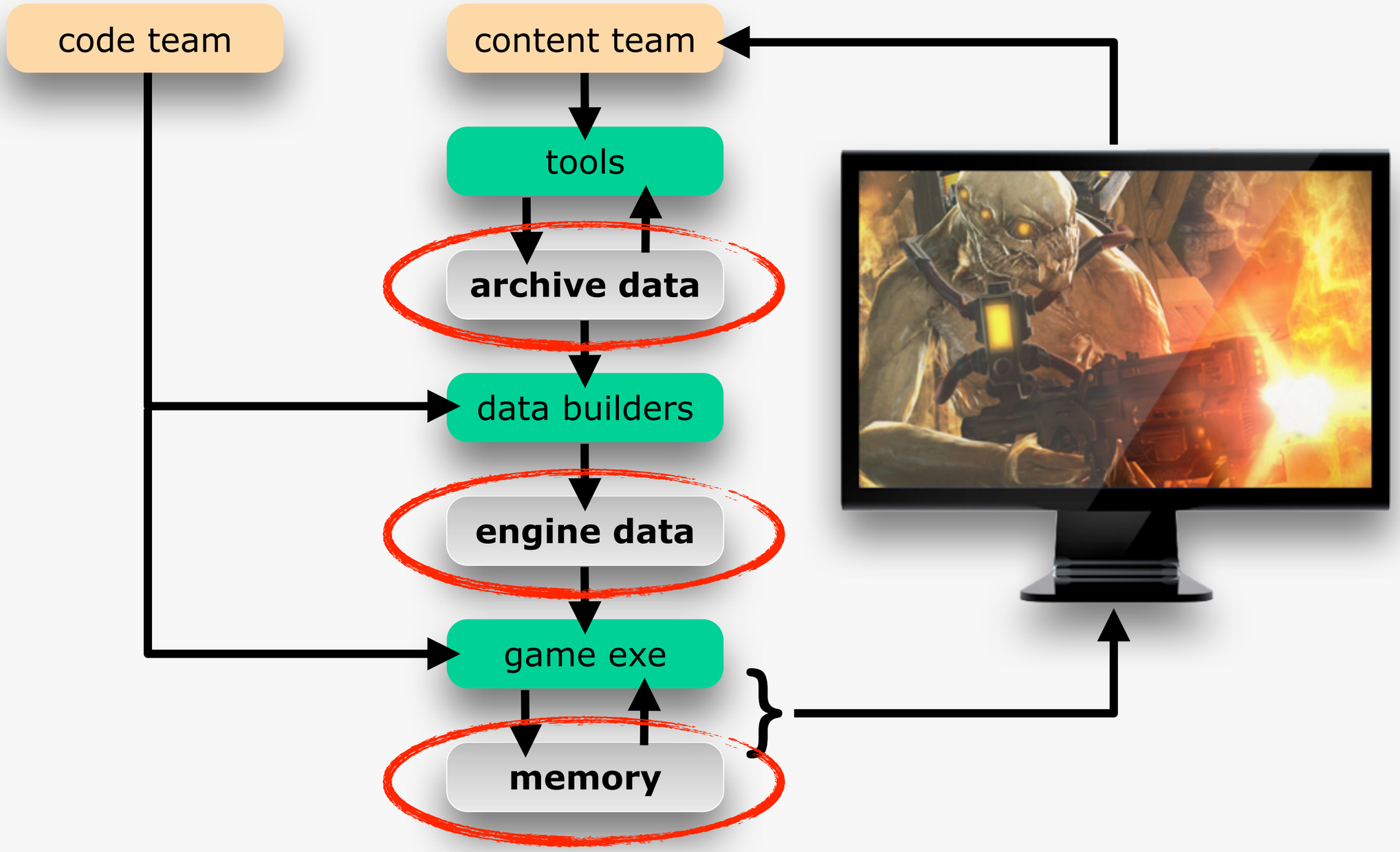
- **"Load-n-Go"**
- "Read-n-Build"
- "Structured Binary"

First let me show you what makes LOAD-N-GO so efficient, and horrible for your production pipeline.



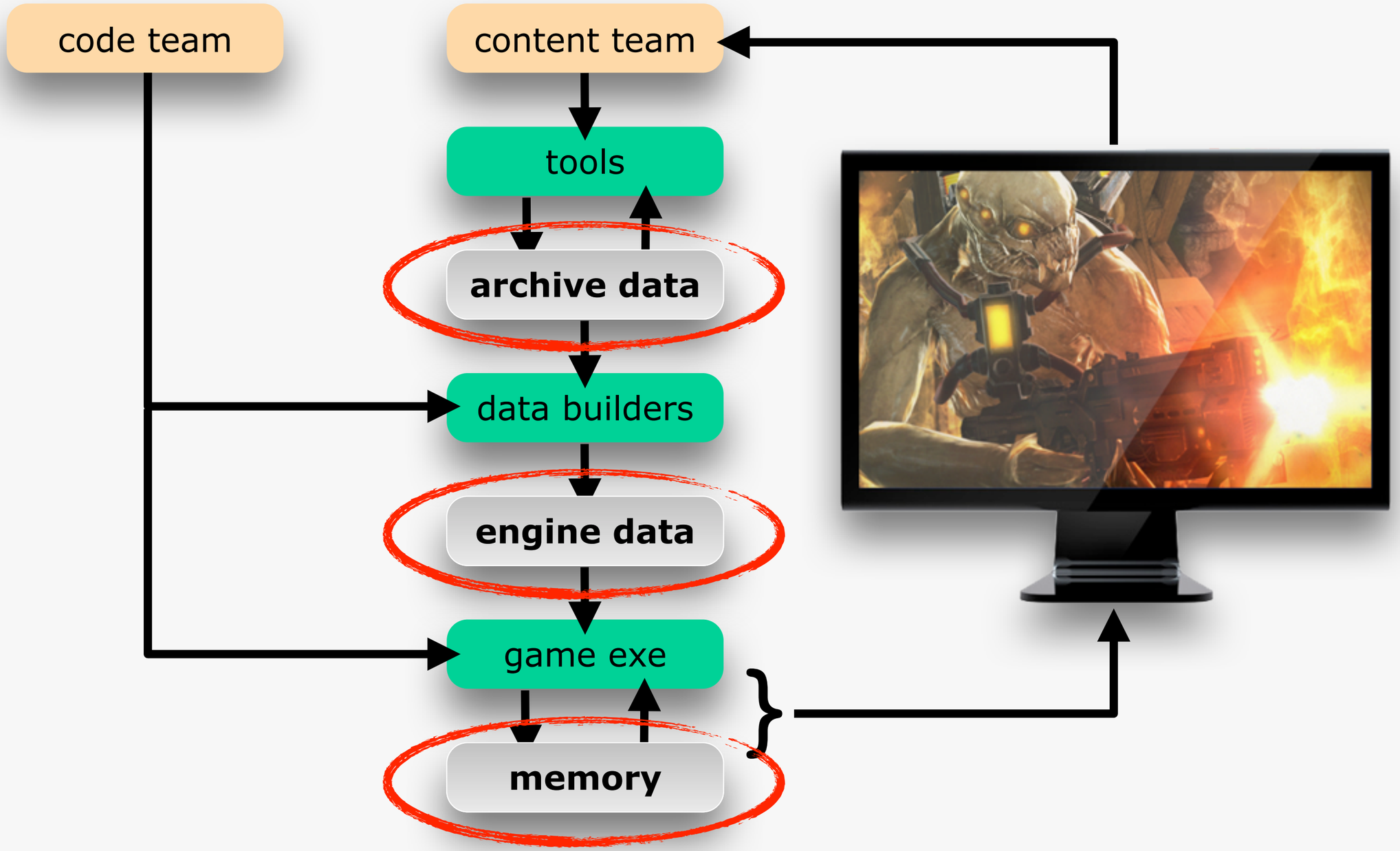
Over next slides, I'll be talking about archive files, engine files, and memory

Keep an eye on where they are in the production pipeline, while I do my Magic Move slide transition. Thank you Steve Jobs



Over next slides, I'll be talking about archive files, engine files, and memory

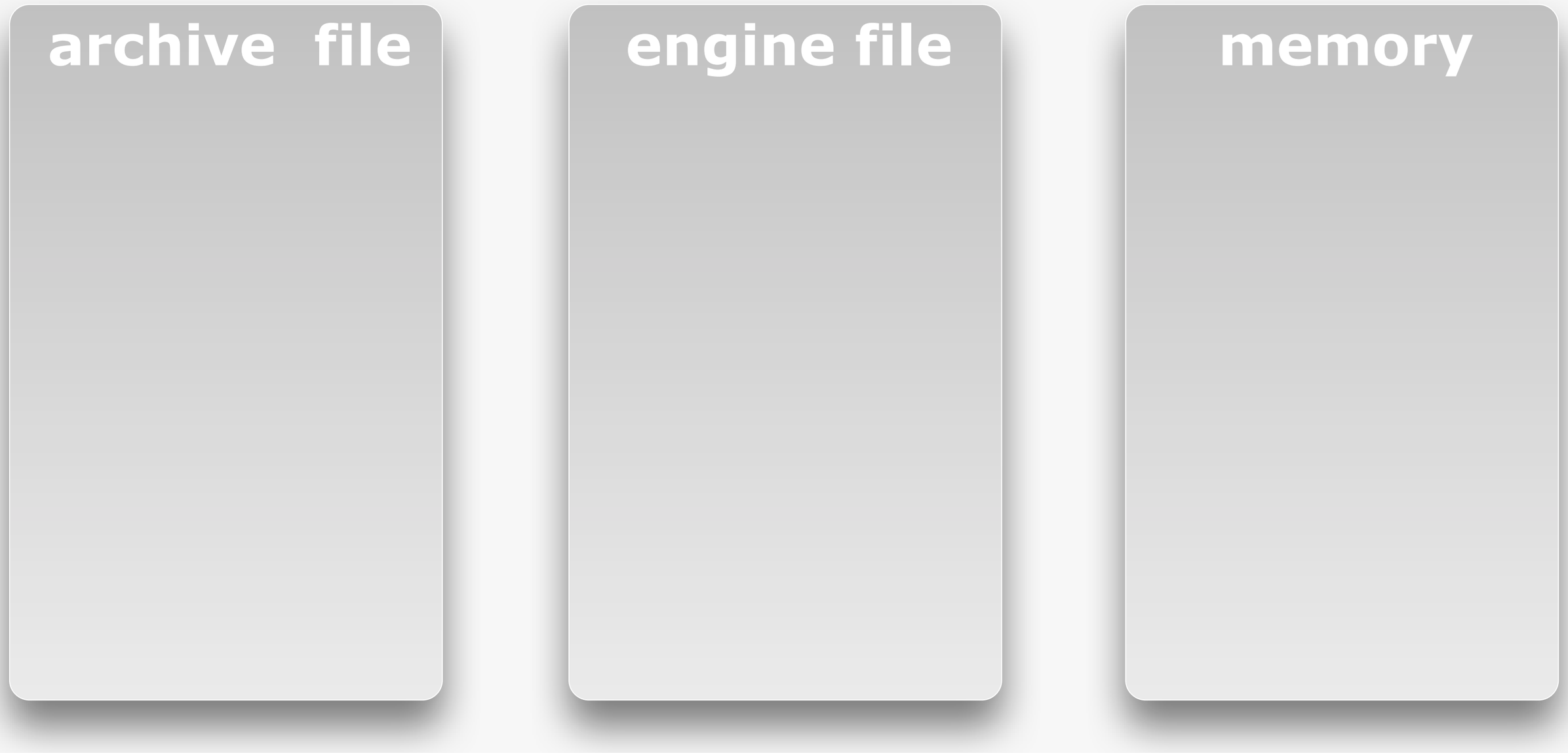
Keep an eye on where they are in the production pipeline, while I do my Magic Move slide transition. Thank you Steve Jobs


code team
content team
tools
archive data
data builders
engine data
game exe
memory

A data builder reads the wordy archive file, and fits everything in the right places in the engine file

A data builder reads the wordy archive file, and fits everything in the right places in the engine file

“Load-n-Go” data files



[CLICK] So on the left we have an archive file. Don't worry if you can't read it. I realize that the text is a little small. It's a random piece of XML. The text is not important, it's just to show that there's a lot of stuff in there that we don't necessarily want to be parsing at runtime.

[CLICK] And on the right, a C struct. Again, don't worry if you cant read. It's not an exciting read.
And remember I’m demonstrating a data loading principle, not a specific file format.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

[CLICK] So on the left we have an archive file. Don't worry if you can't read it. I realize that the text is a little small. It's a random piece of XML. The text is not important, it's just to show that there's a lot of stuff in there that we don't necessarily want to be parsing at runtime.

[CLICK] And on the right, a C struct. Again, don't worry if you cant read. It's not an exciting read.
And remember I’m demonstrating a data loading principle, not a specific file format.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

[CLICK] So on the left we have an archive file. Don't worry if you can't read it. I realize that the text is a little small. It's a random piece of XML. The text is not important, it's just to show that there's a lot of stuff in there that we don't necessarily want to be parsing at runtime.

[CLICK] And on the right, a C struct. Again, don't worry if you cant read. It's not an exciting read.
And remember I’m demonstrating a data loading principle, not a specific file format.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

In a LOAD-N-GO data loading system, the format of the engine file is byte-for-byte identical with the C struct that it will be used to initialize.

"Load-n-Go" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

In a LOAD-N-GO data loading system, the format of the engine file is byte-for-byte identical with the C struct that it will be used to initialize.

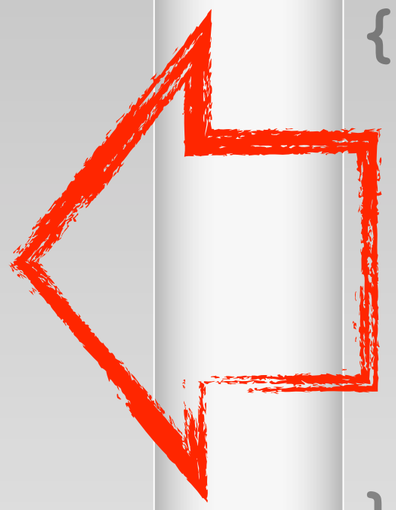
"Load-n-Go" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```



memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

In a LOAD-N-GO data loading system, the format of the engine file is byte-for-byte identical with the C struct that it will be used to initialize.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data builder needs to be compiled with the target C structure.

[CLICK] When the builder runs, it will parse the archive format, place values in the various struct fields, and then write the file to disk as a binary block.

"Load-n-Go" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

data builder

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data builder needs to be compiled with the target C structure.

[CLICK] When the builder runs, it will parse the archive format, place values in the various struct fields, and then write the file to disk as a binary block.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

256
192
100.0
200.0
300.0

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data builder needs to be compiled with the target C structure.

[CLICK] When the builder runs, it will parse the archive format, place values in the various struct fields, and then write the file to disk as a binary block.

"Load-n-Go" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data format is not in any way expressed in the file itself. It is a blind block of raw data.

[CLICK] It's a mystery box.
The only way to use this data, is to initialize the matching runtime C struct with it. In order to make sure that the data in the file and the C struct match, is to mark it with a version number. That information is stored in a special portion of the file.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data format is not in any way expressed in the file itself. It is a blind block of raw data.

[CLICK] It's a mystery box.
The only way to use this data, is to initialize the matching runtime C struct with it. In order to make sure that the data in the file and the C struct match, is to mark it with a version number. That information is stored in a special portion of the file.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data format is not in any way expressed in the file itself. It is a blind block of raw data.

[CLICK] It's a mystery box.
The only way to use this data, is to initialize the matching runtime C struct with it. In order to make sure that the data in the file and the C struct match, is to mark it with a version number. That information is stored in a special portion of the file.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

When it is time for the game to load this file, it must first verify that the data in the file matches the current C struct. So it compares the version number of the file, with the version number it is expecting.

"Load-n-Go" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

runtime loader

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

When it is time for the game to load this file, it must first verify that the data in the file matches the current C struct. So it compares the version number of the file, with the version number it is expecting.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

runtime loader

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

When it is time for the game to load this file, it must first verify that the data in the file matches the current C struct. So it compares the version number of the file, with the version number it is expecting.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

runtime loader

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

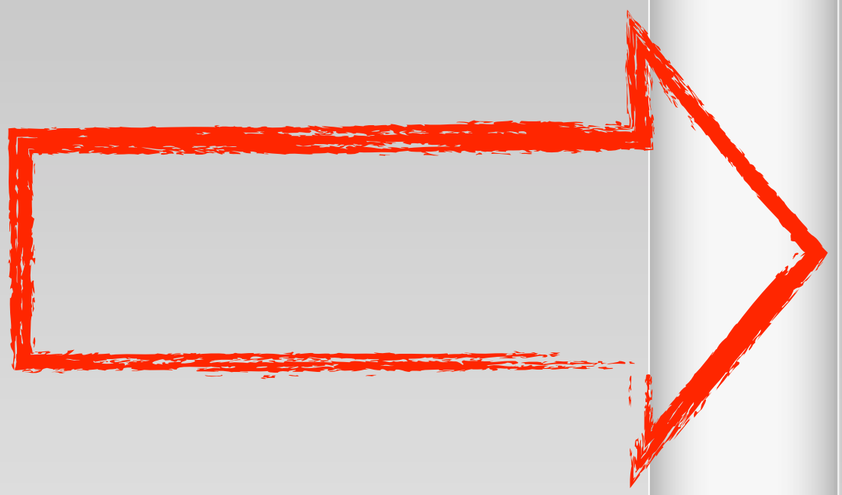
If all is clear, the binary block from the file is written into memory, pointers are set up, and the data is ready to go.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

If all is clear, the binary block from the file is written into memory, pointers are set up, and the data is ready to go.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

Whoa! That was fast!

I can’t imagine a faster way to get your data from disk into your program, ready for use.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

256
192
100.0
200.0
300.0

Whoa! That was fast!

I can’t imagine a faster way to get your data from disk into your program, ready for use.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

But here’s the rub.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;

 float x;
 float y;
 float z;
};
```

Tomorrow we may change the C struct.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

Tomorrow we may change the C struct.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

And if you change the C struct by even a hair, it needs to be marked with a new version number.
Now the runtime will detect a discrepancy, and prohibits the load.

[CLICK] And now the engine file is useless. There is no way to use the file anymore. The old C struct was the only way to properly access the data in the file, and it is gone.

“Load-n-Go” data files

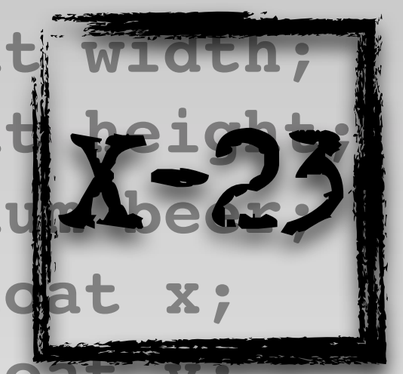
archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```



And if you change the C struct by even a hair, it needs to be marked with a new version number.
Now the runtime will detect a discrepancy, and prohibits the load.

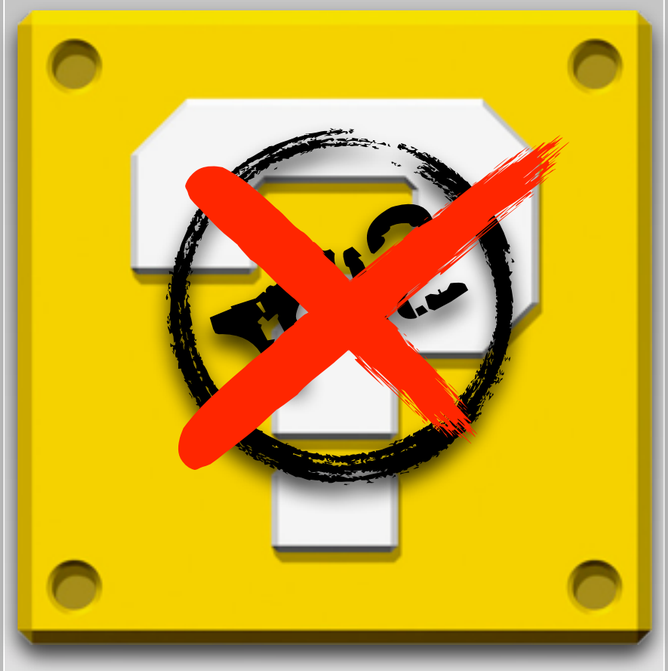
[CLICK] And now the engine file is useless. There is no way to use the file anymore. The old C struct was the only way to properly access the data in the file, and it is gone.

"Load-n-Go" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



runtime loader

memory

```
struct
{
 int width;
 int height;
 enum type;
 float x;
 float y;
 float z;
};
```

And if you change the C struct by even a hair, it needs to be marked with a new version number.
Now the runtime will detect a discrepancy, and prohibits the load.

[CLICK] And now the engine file is useless. There is no way to use the file anymore. The old C struct was the only way to properly access the data in the file, and it is gone.

"Load-n-Go" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

And if you change the C struct by even a hair, it needs to be marked with a new version number.
Now the runtime will detect a discrepancy, and prohibits the load.

[CLICK] And now the engine file is useless. There is no way to use the file anymore. The old C struct was the only way to properly access the data in the file, and it is gone.

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

So now we must compile the data builder with the new target C struct, and rebuild all engine files.

Which would not be much of an issue, if it weren’t for the fact...

“Load-n-Go” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

So now we must compile the data builder with the new target C struct, and rebuild all engine files.

Which would not be much of an issue, if it weren’t for the fact...

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

...that there's a 100,000 of them.

...that there's a 100,000 of them.

“Load-n-Go” data files

So LOAD-N-GO is the quickest way to get your data from disk to memory and usable. There is hardly any CPU processing involved. Perhaps a few pointer fix-ups, but that’s very minor.

[CLICK] LOAD-N-GO has another advantage, that makes it very attractive for streaming open world games. Because the data on disk is exactly the same size as it will take in memory, you can load it directly into the block of memory where it will live for its entire life span. There is no need to allocate larger intermediate buffers. The Saboteur used a fixed pool of fixed size buffers.
[CLICK][CLICK]

“Load-n-Go” data files

- Pro: Minimal CPU overhead

So LOAD-N-GO is the quickest way to get your data from disk to memory and usable. There is hardly any CPU processing involved. Perhaps a few pointer fix-ups, but that’s very minor.

[CLICK] LOAD-N-GO has another advantage, that makes it very attractive for streaming open world games. Because the data on disk is exactly the same size as it will take in memory, you can load it directly into the block of memory where it will live for its entire life span. There is no need to allocate larger intermediate buffers. The Saboteur used a fixed pool of fixed size buffers.
[CLICK][CLICK]

"Load-n-Go" data files

- Pro: Minimal CPU overhead
- Pro: Simple memory management

So LOAD-N-GO is the quickest way to get your data from disk to memory and usable. There is hardly any CPU processing involved. Perhaps a few pointer fix-ups, but that's very minor.

[CLICK] LOAD-N-GO has another advantage, that makes it very attractive for streaming open world games. Because the data on disk is exactly the same size as it will take in memory, you can load it directly into the block of memory where it will live for its entire life span. There is no need to allocate larger intermediate buffers. The Saboteur used a fixed pool of fixed size buffers.
[CLICK][CLICK]

"Load-n-Go" data files

- Pro: Minimal CPU overhead
- Pro: Simple memory management
- Con: Strong data/code dependency means frequent long data rebuilds

So LOAD-N-GO is the quickest way to get your data from disk to memory and usable. There is hardly any CPU processing involved. Perhaps a few pointer fix-ups, but that's very minor.

[CLICK] LOAD-N-GO has another advantage, that makes it very attractive for streaming open world games. Because the data on disk is exactly the same size as it will take in memory, you can load it directly into the block of memory where it will live for its entire life span. There is no need to allocate larger intermediate buffers. The Saboteur used a fixed pool of fixed size buffers.
[CLICK][CLICK]

"Load-n-Go" data files

- Pro: Minimal CPU overhead
- Pro: Simple memory management
- Con: Strong data/code dependency means frequent long data rebuilds
- Con: Switching to previous build can be time consuming

So LOAD-N-GO is the quickest way to get your data from disk to memory and usable. There is hardly any CPU processing involved. Perhaps a few pointer fix-ups, but that's very minor.

[CLICK] LOAD-N-GO has another advantage, that makes it very attractive for streaming open world games. Because the data on disk is exactly the same size as it will take in memory, you can load it directly into the block of memory where it will live for its entire life span. There is no need to allocate larger intermediate buffers. The Saboteur used a fixed pool of fixed size buffers.
[CLICK][CLICK]

Game data format categories

That was the principle used in The Saboteur. Mercenaries used a technique that I have dubbed READ-N-BUILD.

Game data format categories

- "Load-n-Go"
- **"Read-n-Build"**
- "Structured Binary"

That was the principle used in The Saboteur. Mercenaries used a technique that I have dubbed READ-N-BUILD.

“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

Remember I’m describing a principle here, not a specific format.

[CLICK] Mercenaries used a chunk based format, similar to IFF.

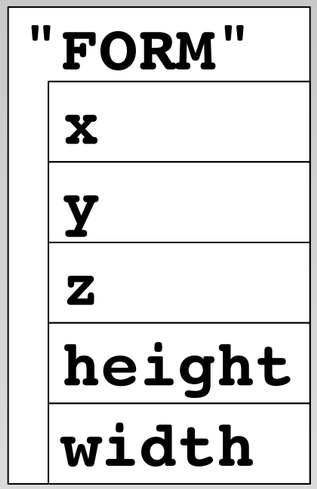
In a chunk based format, such as IFF, fields are laid out in a fixed, stable, known order. They may or may not match a particular C struct. In this case, they don’t. It is the known order of the fields that is important.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

Remember I'm describing a principle here, not a specific format.

[CLICK] Mercenaries used a chunk based format, similar to IFF.

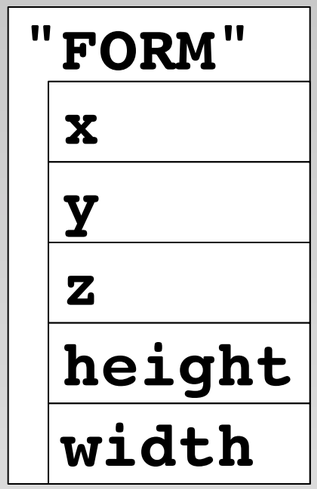
In a chunk based format, such as IFF, fields are laid out in a fixed, stable, known order. They may or may not match a particular C struct. In this case, they don't. It is the known order of the fields that is important.

“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data builder is very similar, except that this time, additional formatting information is included, such as chunk headers. This will help the runtime to navigate the data in the file.

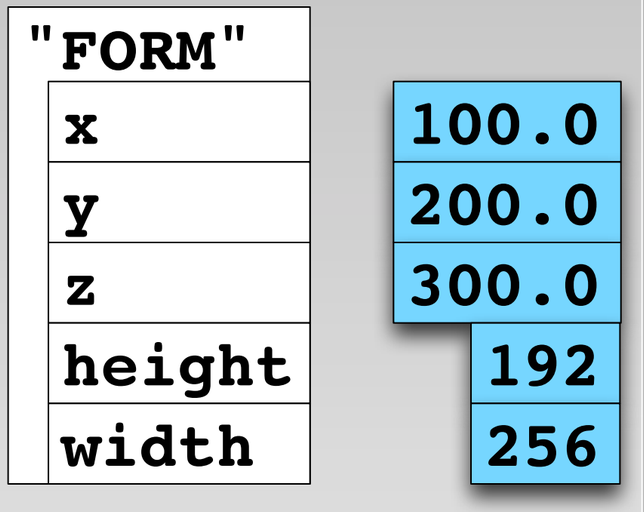
[CLICK] So at runtime, after identifying the chunk and all, we rely on the known fixed order of the data in the file. Like so.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data builder is very similar, except that this time, additional formatting information is included, such as chunk headers. This will help the runtime to navigate the data in the file.

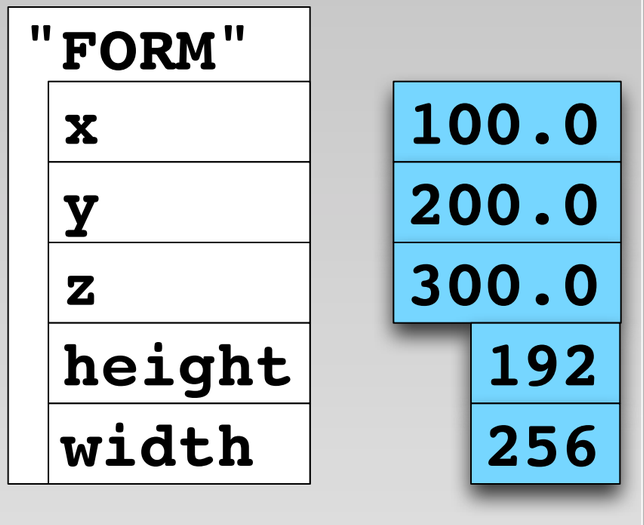
[CLICK] So at runtime, after identifying the chunk and all, we rely on the known fixed order of the data in the file. Like so.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



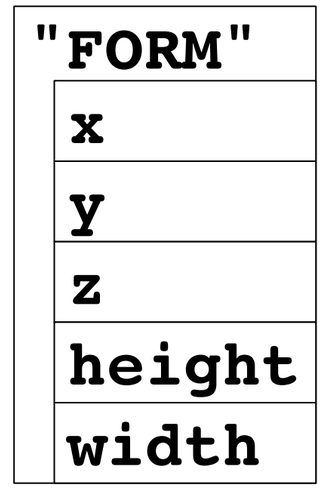
memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data builder is very similar, except that this time, additional formatting information is included, such as chunk headers. This will help the runtime to navigate the data in the file.

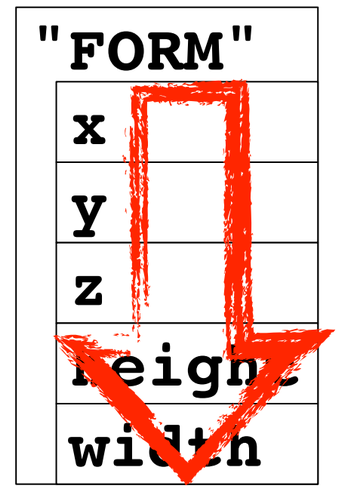
[CLICK] So at runtime, after identifying the chunk and all, we rely on the known fixed order of the data in the file. Like so.

```
void ReadFoo( Foo* foo, IffChunk* chunk )
{
   foo->x      = chunk->NextFloat();
   foo->y      = chunk->NextFloat();
   foo->z      = chunk->NextFloat();
   foo->height = chunk->NextInt();
   foo->width  = chunk->NextInt();
}
```



The reader reads through the fields, one by one, and writes the data in the runtime structure, one field at a time.

```
void ReadFoo( Foo* foo, IffChunk* chunk )
{
   foo->x      = chunk->NextFloat();
   foo->y      = chunk->NextFloat();
   foo->z      = chunk->NextFloat();
   foo->height = chunk->NextInt();
   foo->width  = chunk->NextInt();
}
```



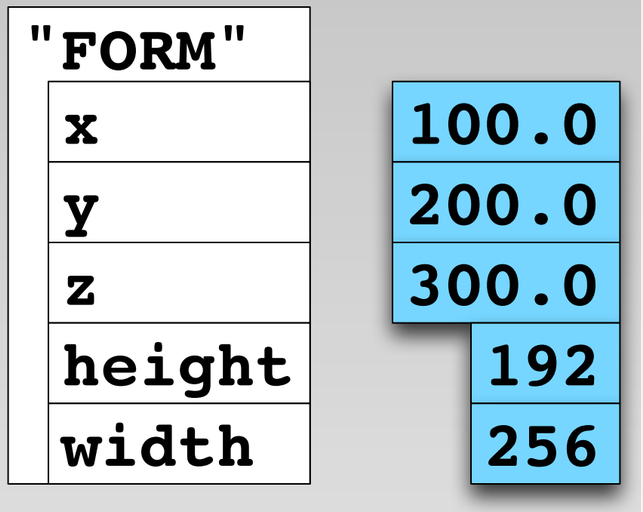
The reader reads through the fields, one by one, and writes the data in the runtime structure, one field at a time.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

[CLICK] And before we do that, we must initialize the runtime struct with defaults.

[CLICK] Then we read the fields.

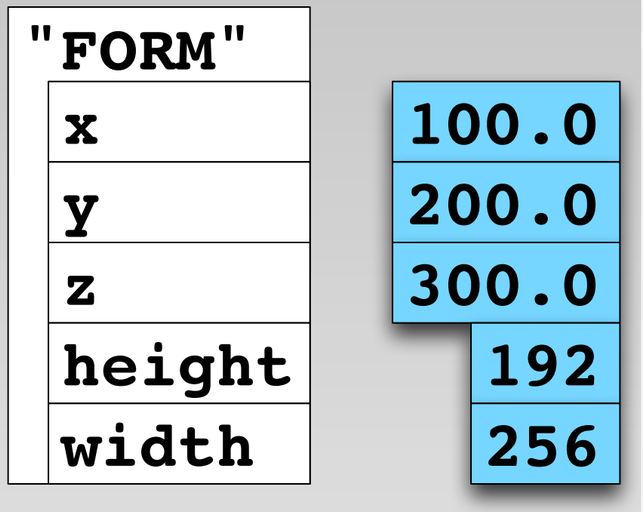
[CLICK]

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;    default
 int height;   default
 float x;      default
 float y;      default
 float z;      default
};
```

[CLICK] And before we do that, we must initialize the runtime struct with defaults.

[CLICK] Then we read the fields.

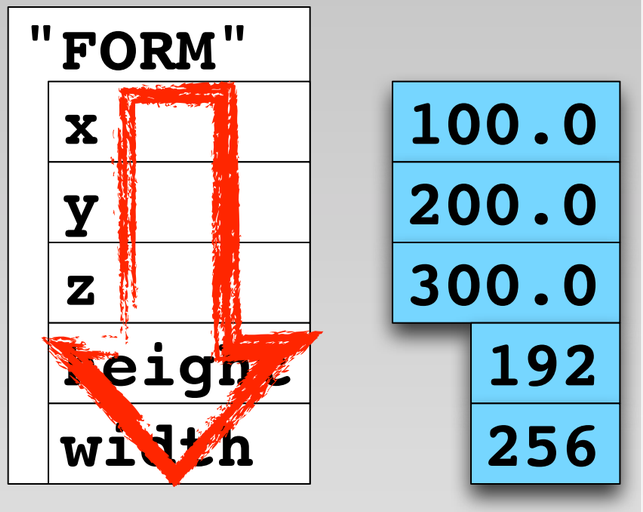
[CLICK]

“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



runtime loader

memory

```
struct
{
 int width;    default
 int height;   default
 float x;      default
 float y;      default
 float z;      default
};
```

[CLICK] And before we do that, we must initialize the runtime struct with defaults.

[CLICK] Then we read the fields.

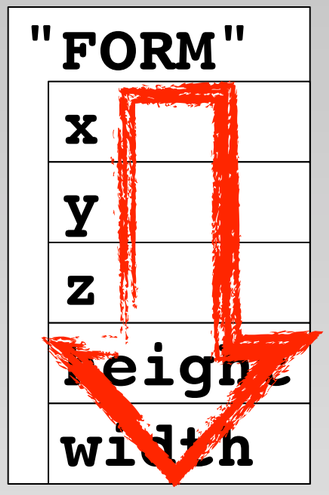
[CLICK]

“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



runtime loader

memory

```
struct
{
 int width;  256
 int height; 192
 float x;    100.0
 float y;    200.0
 float z;    300.0
};
```

[CLICK] And before we do that, we must initialize the runtime struct with defaults.

[CLICK] Then we read the fields.

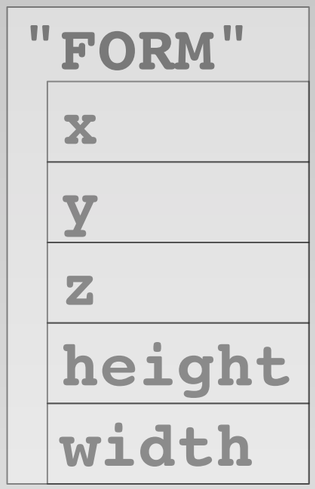
[CLICK]

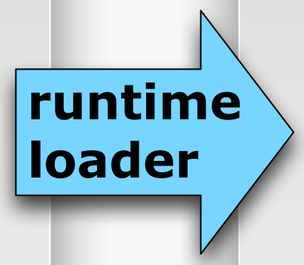
"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file





memory

```
struct
{
 int width;  256
 int height; 192
 float x;    100.0
 float y;    200.0
 float z;    300.0
};
```

[CLICK] And before we do that, we must initialize the runtime struct with defaults.

[CLICK] Then we read the fields.

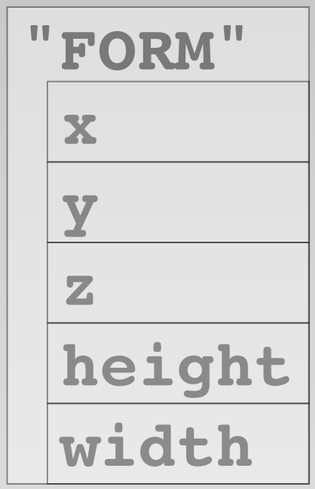
[CLICK]

“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;  256
 int height; 192
 float x;    100.0
 float y;    200.0
 float z;    300.0
};
```

[CLICK] And before we do that, we must initialize the runtime struct with defaults.

[CLICK] Then we read the fields.

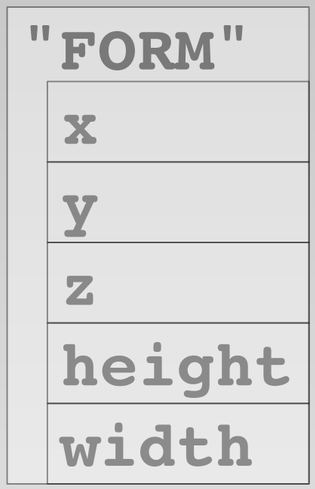
[CLICK]

“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

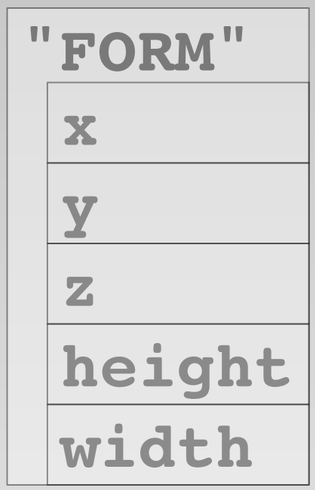
So now the part where LOAD-N-GO had to give up. A change in the runtime format.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;
 int height;

 float x;
 float y;
 float z;
};
```

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn't changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday's data file will be left at the default value in the runtime memory structure.

Of course this doesn't cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

ine file

"FORM"
x
y
z
height
width

today

emory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn't changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday's data file will be left at the default value in the runtime memory structure.

Of course this doesn't cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

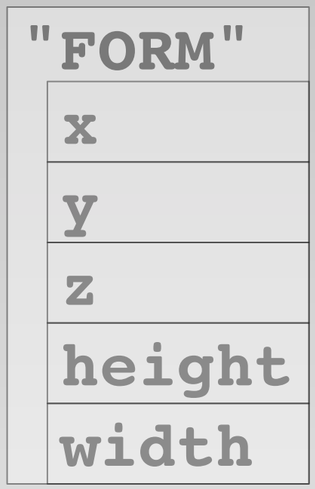
“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

ine file



today

emory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn’t changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday’s data file will be left at the default value in the runtime memory structure.

Of course this doesn’t cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

ine file

"FORM"	
x	100.0
y	200.0
z	300.0
height	192
width	256

runtime loader

today

emory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn't changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday's data file will be left at the default value in the runtime memory structure.

Of course this doesn't cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

ine file

"FORM"	
x	100.0
y	200.0
z	300.0
height	192
width	256

runtime loader

today

emory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn't changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday's data file will be left at the default value in the runtime memory structure.

Of course this doesn't cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

ine file

"FORM"	
x	100.0
y	200.0
z	300.0
height	192
width	256

runtime loader

today

emory

```
struct
{
 int width;    default
 int height;   default
 enum beer;    default
 float x;      default
 float y;      default
 float z;      default
};
```

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn't changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday's data file will be left at the default value in the runtime memory structure.

Of course this doesn't cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

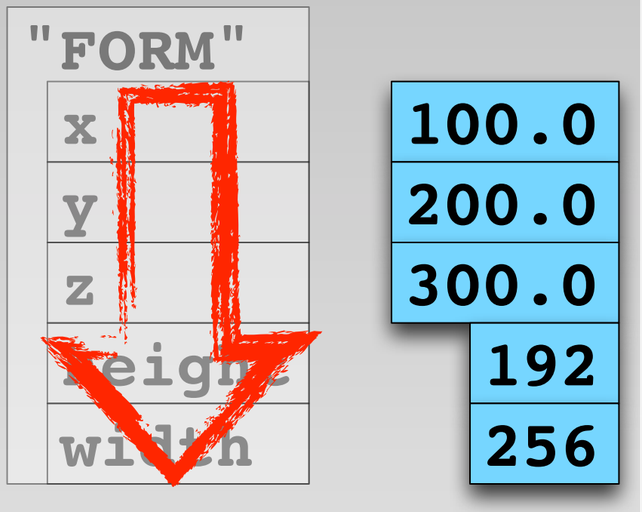
"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

...gine file



today

...emory

```
struct
{
 int width;    default
 int height;   default
 enum beer;    default
 float x;      default
 float y;      default
 float z;      default
};
```

runtime loader

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn't changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday's data file will be left at the default value in the runtime memory structure.

Of course this doesn't cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

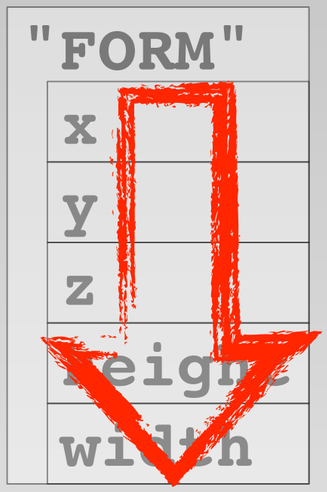
“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

...gine file



today

...emory

```
struct
{
 int width;   256
 int height;  192
 enum beer;   default
 float x;     100.0
 float y;     200.0
 float z;     300.0
};
```

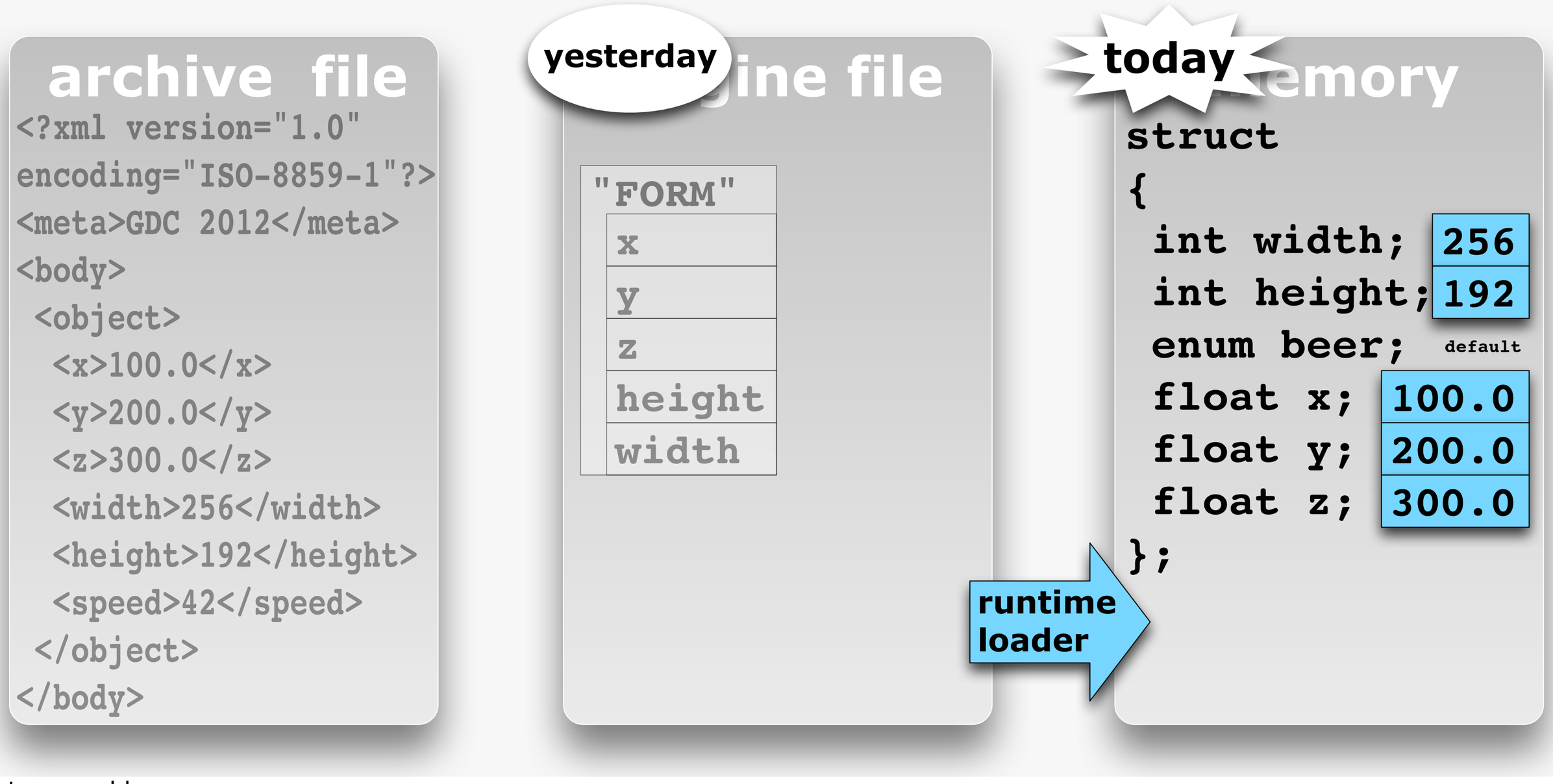
runtime loader

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn’t changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday’s data file will be left at the default value in the runtime memory structure.

Of course this doesn’t cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

"Read-n-Build" data files



This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn't changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday's data file will be left at the default value in the runtime memory structure.

Of course this doesn't cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

yesterday

ine file

"FORM"
x
y
z
height
width

today

emory

```
struct
{
 int width;   256
 int height;  192
 enum beer;   default
 float x;     100.0
 float y;     200.0
 float z;     300.0
};
```

This is no problem.

[CLICK] We can read the data file that was built yesterday with the new reader.
[CLICK] The order of the data in the chunk hasn’t changed.
[CLICK] The reader can determine the length of the chunk from the chunk header, so when it reaches the end of the chunk data, it stops reading.
[CLICK] Any fields that were not in yesterday’s data file will be left at the default value in the runtime memory structure.

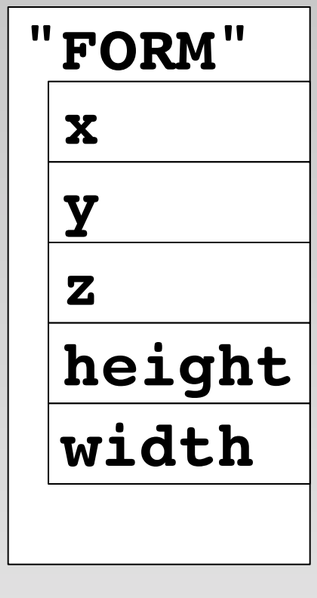
Of course this doesn’t cover every possible data change. Sometimes this simple reordering is not possible. But most of the time, this will cover the 80% case.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

And the other way around is not a problem either.

[CLICK] If you have built data with the new field in it, you can still read it with yesterday's executable. Remember this was important when today's build is broken.
[CLICK] The old reader will only read the fields that it knows about, and ignore additional fields in the chunk.

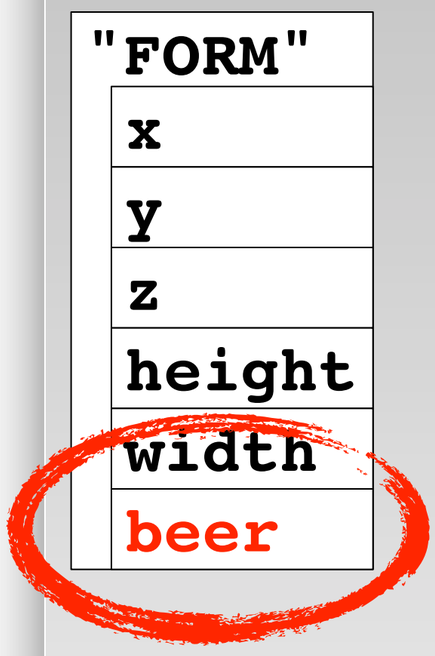
"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

today

gine file



yesterday

emory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

And the other way around is not a problem either.

[CLICK] If you have built data with the new field in it, you can still read it with yesterday's executable. Remember this was important when today's build is broken.
[CLICK] The old reader will only read the fields that it knows about, and ignore additional fields in the chunk.

"Read-n-Build" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

today

...gine file

"FORM"
x
y
z
height
width
beer

yesterday

...emory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

And the other way around is not a problem either.

[CLICK] If you have built data with the new field in it, you can still read it with yesterday's executable. Remember this was important when today's build is broken.
[CLICK] The old reader will only read the fields that it knows about, and ignore additional fields in the chunk.

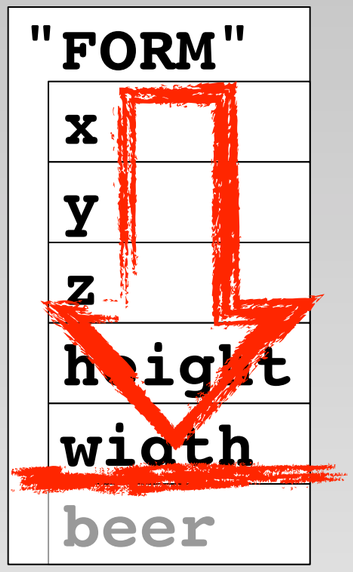
“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

today

engine file



yesterday

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

And the other way around is not a problem either.

[CLICK] If you have built data with the new field in it, you can still read it with yesterday’s executable. Remember this was important when today’s build is broken.
[CLICK] The old reader will only read the fields that it knows about, and ignore additional fields in the chunk.

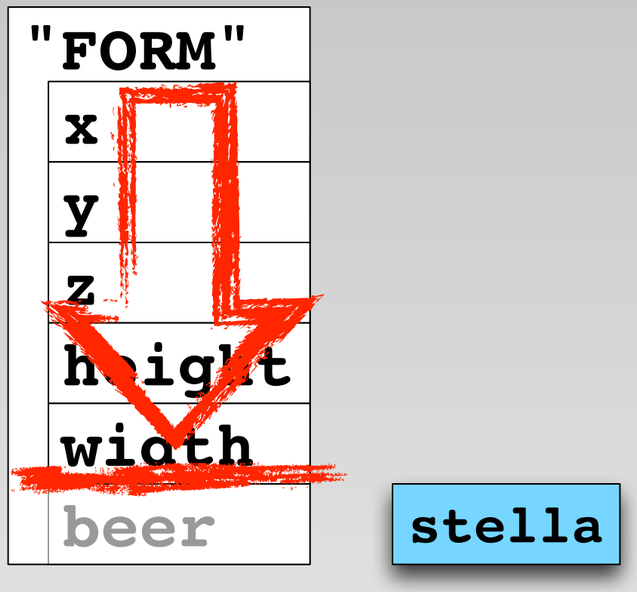
“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

today

...gine file



yesterday

...emory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

256
192
100.0
200.0
300.0

And the other way around is not a problem either.

[CLICK] If you have built data with the new field in it, you can still read it with yesterday’s executable. Remember this was important when today’s build is broken.
[CLICK] The old reader will only read the fields that it knows about, and ignore additional fields in the chunk.

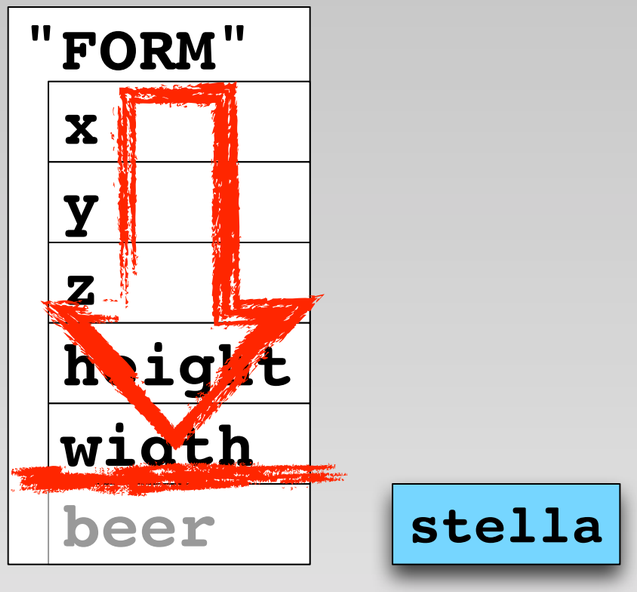
“Read-n-Build” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

today

...gine file



yesterday

...emory

```
struct
{
 int width;   256
 int height;  192
 float x;     100.0
 float y;     200.0
 float z;     300.0
};
```

And the other way around is not a problem either.

[CLICK] If you have built data with the new field in it, you can still read it with yesterday’s executable. Remember this was important when today’s build is broken.
[CLICK] The old reader will only read the fields that it knows about, and ignore additional fields in the chunk.

“Read-n-Build” data files

So Mercenaries had reasonably good forward and backward compatibility.
[CLICK] This allowed developers to switch between versions of the code, most of the time without rebuilding data.
[CLICK] But Mercenaries suffered horrible streaming performance. Having to interpret every individual field of data requires extra CPU resources.
[CLICK] And on top of that, it made a mess of memory management. You need a temporary buffer that is a good deal larger than your final memory format. And we’re not dealing with small data files here. And if you want to maximize streaming throughput, you need TWO temporary buffers at the same time. Load into one while you process the other.
[CLICK] And engineers hated it. They needed to write a custom reader for every chunk. To keep compatibility with older data version, they had to maintain multiple versions of these readers, and often didn’t bother,
[CLICK] so compatibility was rarely achieved. And engineers always liked the LOAD-N-GO efficiency better.
[PAUSE] So... there we are... no winner.

"Read-n-Build" data files

- Pro: Backward/forward compatibility means fewer data rebuilds

So Mercenaries had reasonably good forward and backward compatibility.
[CLICK] This allowed developers to switch between versions of the code, most of the time without rebuilding data.
[CLICK] But Mercenaries suffered horrible streaming performance. Having to interpret every individual field of data requires extra CPU resources.
[CLICK] And on top of that, it made a mess of memory management. You need a temporary buffer that is a good deal larger than your final memory format. And we're not dealing with small data files here. And if you want to maximize streaming throughput, you need TWO temporary buffers at the same time. Load into one while you process the other.
[CLICK] And engineers hated it. They needed to write a custom reader for every chunk. To keep compatibility with older data version, they had to maintain multiple versions of these readers, and often didn't bother,
[CLICK] so compatibility was rarely achieved. And engineers always liked the LOAD-N-GO efficiency better.
[PAUSE] So... there we are... no winner.

"Read-n-Build" data files

- Pro: Backward/forward compatibility means fewer data rebuilds
- Pro: Can switch to yesterday's build

So Mercenaries had reasonably good forward and backward compatibility.
[CLICK] This allowed developers to switch between versions of the code, most of the time without rebuilding data.
[CLICK] But Mercenaries suffered horrible streaming performance. Having to interpret every individual field of data requires extra CPU resources.
[CLICK] And on top of that, it made a mess of memory management. You need a temporary buffer that is a good deal larger than your final memory format. And we're not dealing with small data files here. And if you want to maximize streaming throughput, you need TWO temporary buffers at the same time. Load into one while you process the other.
[CLICK] And engineers hated it. They needed to write a custom reader for every chunk. To keep compatibility with older data version, they had to maintain multiple versions of these readers, and often didn't bother,
[CLICK] so compatibility was rarely achieved. And engineers always liked the LOAD-N-GO efficiency better.
[PAUSE] So... there we are... no winner.

"Read-n-Build" data files

- Pro: Backward/forward compatibility means fewer data rebuilds
- Pro: Can switch to yesterday's build
- Con: Considerable CPU overhead

So Mercenaries had reasonably good forward and backward compatibility.
[CLICK] This allowed developers to switch between versions of the code, most of the time without rebuilding data.
[CLICK] But Mercenaries suffered horrible streaming performance. Having to interpret every individual field of data requires extra CPU resources.
[CLICK] And on top of that, it made a mess of memory management. You need a temporary buffer that is a good deal larger than your final memory format. And we're not dealing with small data files here. And if you want to maximize streaming throughput, you need TWO temporary buffers at the same time. Load into one while you process the other.
[CLICK] And engineers hated it. They needed to write a custom reader for every chunk. To keep compatibility with older data version, they had to maintain multiple versions of these readers, and often didn't bother,
[CLICK] so compatibility was rarely achieved. And engineers always liked the LOAD-N-GO efficiency better.
[PAUSE] So... there we are... no winner.

"Read-n-Build" data files

- Pro: Backward/forward compatibility means fewer data rebuilds
- Pro: Can switch to yesterday's build
- Con: Considerable CPU overhead
- Con: Memory management issues

So Mercenaries had reasonably good forward and backward compatibility.
[CLICK] This allowed developers to switch between versions of the code, most of the time without rebuilding data.
[CLICK] But Mercenaries suffered horrible streaming performance. Having to interpret every individual field of data requires extra CPU resources.
[CLICK] And on top of that, it made a mess of memory management. You need a temporary buffer that is a good deal larger than your final memory format. And we're not dealing with small data files here. And if you want to maximize streaming throughput, you need TWO temporary buffers at the same time. Load into one while you process the other.
[CLICK] And engineers hated it. They needed to write a custom reader for every chunk. To keep compatibility with older data version, they had to maintain multiple versions of these readers, and often didn't bother,
[CLICK] so compatibility was rarely achieved. And engineers always liked the LOAD-N-GO efficiency better.
[PAUSE] So... there we are... no winner.

"Read-n-Build" data files

- Pro: Backward/forward compatibility means fewer data rebuilds
- Pro: Can switch to yesterday's build
- Con: Considerable CPU overhead
- Con: Memory management issues
- Con: Engineers hated it

So Mercenaries had reasonably good forward and backward compatibility.
[CLICK] This allowed developers to switch between versions of the code, most of the time without rebuilding data.
[CLICK] But Mercenaries suffered horrible streaming performance. Having to interpret every individual field of data requires extra CPU resources.
[CLICK] And on top of that, it made a mess of memory management. You need a temporary buffer that is a good deal larger than your final memory format. And we're not dealing with small data files here. And if you want to maximize streaming throughput, you need TWO temporary buffers at the same time. Load into one while you process the other.
[CLICK] And engineers hated it. They needed to write a custom reader for every chunk. To keep compatibility with older data version, they had to maintain multiple versions of these readers, and often didn't bother,
[CLICK] so compatibility was rarely achieved. And engineers always liked the LOAD-N-GO efficiency better.
[PAUSE] So... there we are... no winner.

"Read-n-Build" data files

- Pro: Backward/forward compatibility means fewer data rebuilds **(in theory)**
- Pro: Can switch to yesterday's build **(in theory)**
- Con: Considerable CPU overhead
- Con: Memory management issues
- Con: Engineers hated it

So Mercenaries had reasonably good forward and backward compatibility.
[CLICK] This allowed developers to switch between versions of the code, most of the time without rebuilding data.
[CLICK] But Mercenaries suffered horrible streaming performance. Having to interpret every individual field of data requires extra CPU resources.
[CLICK] And on top of that, it made a mess of memory management. You need a temporary buffer that is a good deal larger than your final memory format. And we're not dealing with small data files here. And if you want to maximize streaming throughput, you need TWO temporary buffers at the same time. Load into one while you process the other.
[CLICK] And engineers hated it. They needed to write a custom reader for every chunk. To keep compatibility with older data version, they had to maintain multiple versions of these readers, and often didn't bother,
[CLICK] so compatibility was rarely achieved. And engineers always liked the LOAD-N-GO efficiency better.
[PAUSE] So... there we are... no winner.

The Holy Grail

Now remember that this talk is NOT about finding the most efficient streaming format. It is about keeping team productivity flowing when stuff breaks. For that, you need forward and backward compatibility, so that's what I'm after here. However if the format is so inefficient that no one wants to use it, it's no good either.

So I went looking for a format that had everything.
[CLICK] Compatibility,
[CLICK] as well as efficiency.
[CLICK] And if it is a hassle for engineers to maintain compatibility, that's no winning option either.

The Holy Grail

- Backward and forward compatibility (à la Read-n-Build)

Now remember that this talk is NOT about finding the most efficient streaming format. It is about keeping team productivity flowing when stuff breaks. For that, you need forward and backward compatibility, so that's what I'm after here. However if the format is so inefficient that no one wants to use it, it's no good either.

So I went looking for a format that had everything.
[CLICK] Compatibility,
[CLICK] as well as efficiency.
[CLICK] And if it is a hassle for engineers to maintain compatibility, that's no winning option either.

The Holy Grail

- Backward and forward compatibility (à la Read-n-Build)
- Suitable for binary block loading (à la Load-n-Go)

Now remember that this talk is NOT about finding the most efficient streaming format. It is about keeping team productivity flowing when stuff breaks. For that, you need forward and backward compatibility, so that's what I'm after here. However if the format is so inefficient that no one wants to use it, it's no good either.

So I went looking for a format that had everything.
[CLICK] Compatibility,
[CLICK] as well as efficiency.
[CLICK] And if it is a hassle for engineers to maintain compatibility, that's no winning option either.

The Holy Grail

- Backward and forward compatibility (à la Read-n-Build)
- Suitable for binary block loading (à la Load-n-Go)
- No additional work for engineers

Now remember that this talk is NOT about finding the most efficient streaming format. It is about keeping team productivity flowing when stuff breaks. For that, you need forward and backward compatibility, so that's what I'm after here. However if the format is so inefficient that no one wants to use it, it's no good either.

So I went looking for a format that had everything.
[CLICK] Compatibility,
[CLICK] as well as efficiency.
[CLICK] And if it is a hassle for engineers to maintain compatibility, that's no winning option either.

Game data format categories

So I went looking for the holy grail, and could not find it. I found there is a remarkable shortage of general purpose binary formats. And none that offer the kind of tightly packed binary blocks that I was looking for.

So I made one up.

Game data format categories

- "Load-n-Go"
- "Read-n-Build"
- **"Structured Binary"**

So I went looking for the holy grail, and could not find it. I found there is a remarkable shortage of general purpose binary formats. And none that offer the kind of tightly packed binary blocks that I was looking for.

So I made one up.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

This is how Structured Binary works.

For the most part, it is the same as LOAD-N-GO. So I’ll fast forward.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

This is how Structured Binary works.

For the most part, it is the same as LOAD-N-GO. So I’ll fast forward.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

This is how Structured Binary works.

For the most part, it is the same as LOAD-N-GO. So I’ll fast forward.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

This is how Structured Binary works.

For the most part, it is the same as LOAD-N-GO. So I’ll fast forward.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

This is how Structured Binary works.

For the most part, it is the same as LOAD-N-GO. So I’ll fast forward.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

256
192
100.0
200.0
300.0

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

This is how Structured Binary works.

For the most part, it is the same as LOAD-N-GO. So I'll fast forward.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;  256
 int height; 192
 float x;    100.0
 float y;    200.0
 float z;    300.0
};
```

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

Alright, here we are. Same place as LOAD-N-GO, with a freshly built block of mystery data. And here there is an extra step.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

Alright, here we are. Same place as LOAD-N-GO, with a freshly built block of mystery data. And here there is an extra step.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

We store a copy of the C-struct. The entire C-struct is tokenized, and stored in the file along with the mystery data.

“Structured Binary” data files

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memor

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

We store a copy of the C-struct. The entire C-struct is tokenized, and stored in the file along with the mystery data.

“Structured Binary” data files

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memor

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

We store a copy of the C-struct. The entire C-struct is tokenized, and stored in the file along with the mystery data.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

We store a copy of the C-struct. The entire C-struct is tokenized, and stored in the file along with the mystery data.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

Runtime loading is exactly the same as LOAD-N-GO. A version number is stored in the file, so we know it is compatible with our current set of C structs.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

Runtime loading is exactly the same as LOAD-N-GO. A version number is stored in the file, so we know it is compatible with our current set of C structs.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

If the version number matches, we load, and we go.

This is exactly the same, and just as fast as LOAD-N-GO

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

runtime loader

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

If the version number matches, we load, and we go.

This is exactly the same, and just as fast as LOAD-N-GO

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

runtime loader

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

If the version number matches, we load, and we go.

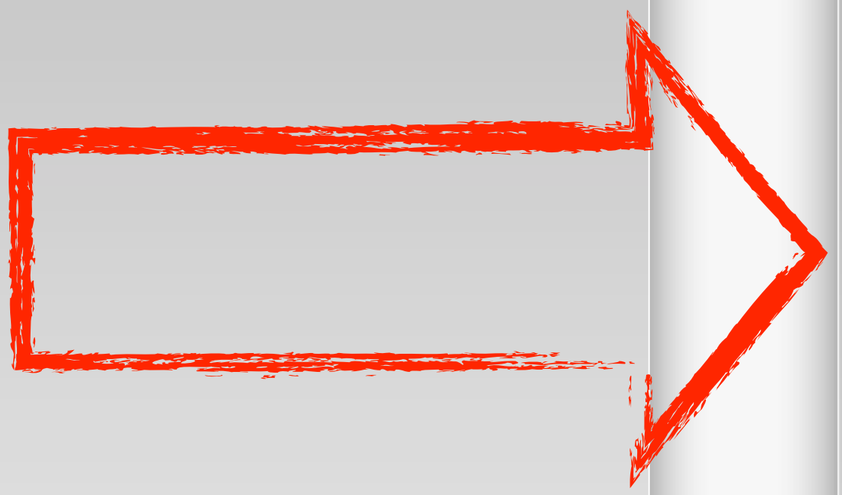
This is exactly the same, and just as fast as LOAD-N-GO

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



memory

If the version number matches, we load, and we go.

This is exactly the same, and just as fast as LOAD-N-GO

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

If the version number matches, we load, and we go.

This is exactly the same, and just as fast as LOAD-N-GO

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

256
192
100.0
200.0
300.0

If the version number matches, we load, and we go.

This is exactly the same, and just as fast as LOAD-N-GO

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

Now this is the part where LOAD-N-GO caused us grief.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;

 float x;
 float y;
 float z;
};
```

A data change!

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

A data change!

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

[CLICK] The runtime loader detects a mismatch.
But this time, instead of scrapping the data file and starting from scratch, we have a fall-back option.
Remember, we stored the schema, the C-struct definition right in the data file.

“Structured Binary” data files

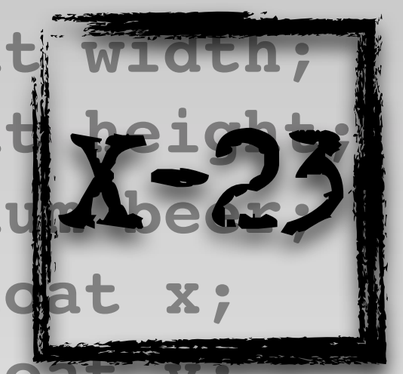
archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```



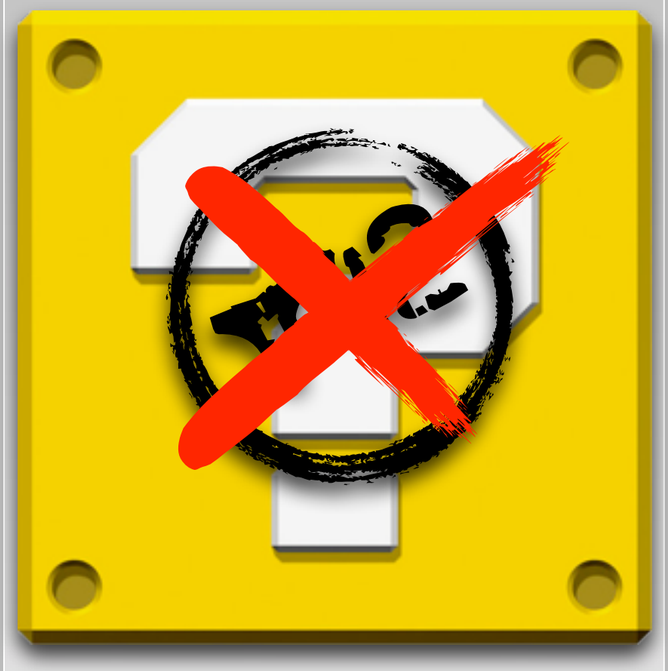
[CLICK] The runtime loader detects a mismatch.
But this time, instead of scrapping the data file and starting from scratch, we have a fall-back option.
Remember, we stored the schema, the C-struct definition right in the data file.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file



runtime loader

memory

```
struct
{
 int width;
 int height;
 enum kind;
 float x;
 float y;
 float z;
};
```

[CLICK] The runtime loader detects a mismatch.
But this time, instead of scrapping the data file and starting from scratch, we have a fall-back option.
Remember, we stored the schema, the C-struct definition right in the data file.

“Structured Binary” data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

runtime loader

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

Let’s take a closer look.

"Structured Binary" data file

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

en ile

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

runtime loader

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

Let's take a closer look.

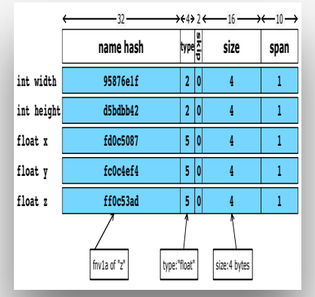
```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

This is a tokenized, compact binary encoding of the C struct, the schema, at the time of creation. This is all we need to know to take the data in the file apart.

A minute ago, I promised a format that doesn't require extra work for the programmers. Where does this extra data come from?

A Insomniac, we have been using a Data Definition Language for quite a while. Every C structure that needs to be serialized, is written in DDL instead. This very much like a C struct definition.
Then we have a data compiler that outputs an equivalent C++ header, and a C++ file with with generated code for accessors and serializers. The same DDL compiler can output this schema.

In the experimental version that I have running currently, the schema has to be hand made. But once we automate that, the engineers need to do nothing extra. They just continue to use DDL to define their data structures.

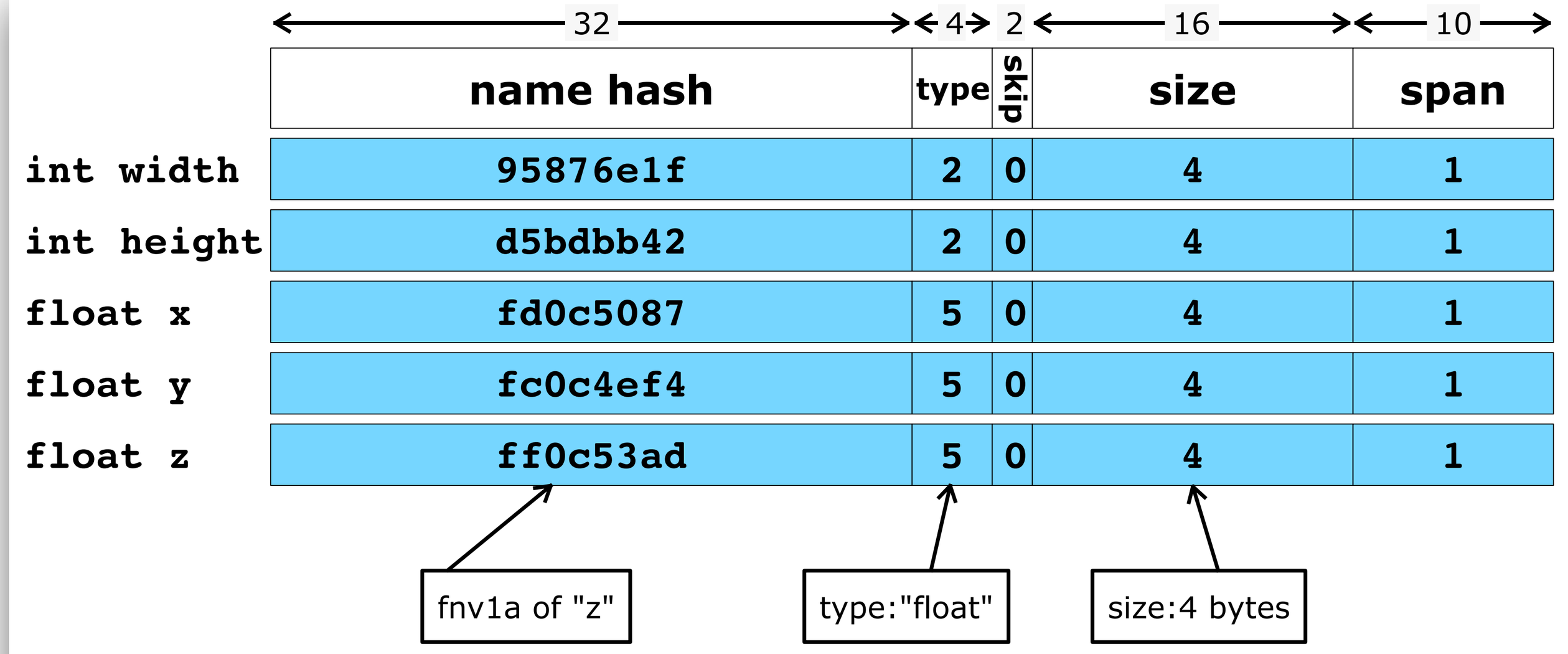


This is a tokenized, compact binary encoding of the C struct, the schema, at the time of creation. This is all we need to know to take the data in the file apart.

A minute ago, I promised a format that doesn't require extra work for the programmers. Where does this extra data come from?

A Insomniac, we have been using a Data Definition Language for quite a while. Every C structure that needs to be serialized, is written in DDL instead. This very much like a C struct definition.
Then we have a data compiler that outputs an equivalent C++ header, and a C++ file with with generated code for accessors and serializers. The same DDL compiler can output this schema.

In the experimental version that I have running currently, the schema has to be hand made. But once we automate that, the engineers need to do nothing extra. They just continue to use DDL to define their data structures.

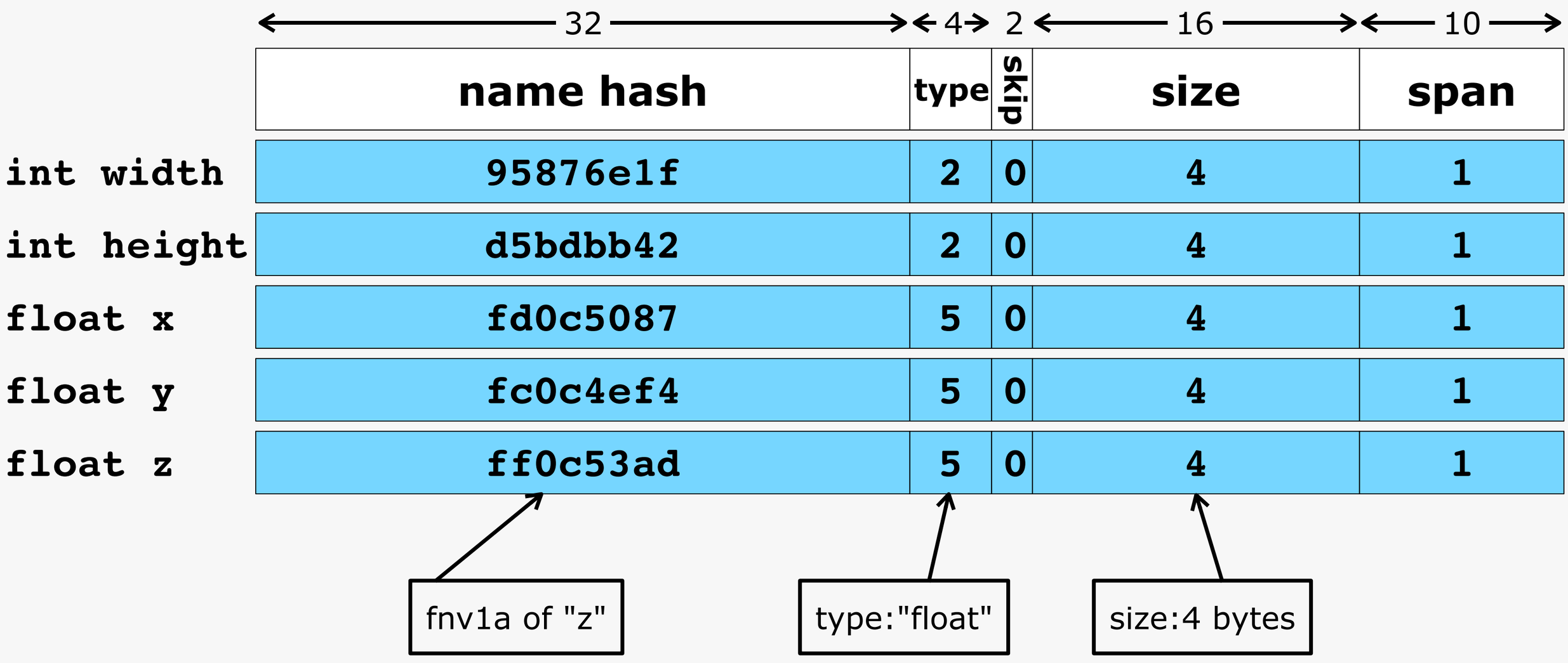


This is a tokenized, compact binary encoding of the C struct, the schema, at the time of creation. This is all we need to know to take the data in the file apart.

A minute ago, I promised a format that doesn't require extra work for the programmers. Where does this extra data come from?

A Insomniac, we have been using a Data Definition Language for quite a while. Every C structure that needs to be serialized, is written in DDL instead. This very much like a C struct definition.
Then we have a data compiler that outputs an equivalent C++ header, and a C++ file with with generated code for accessors and serializers.
The same DDL compiler can output this schema.

In the experimental version that I have running currently, the schema has to be hand made. But once we automate that, the engineers need to do nothing extra. They just continue to use DDL to define their data structures.

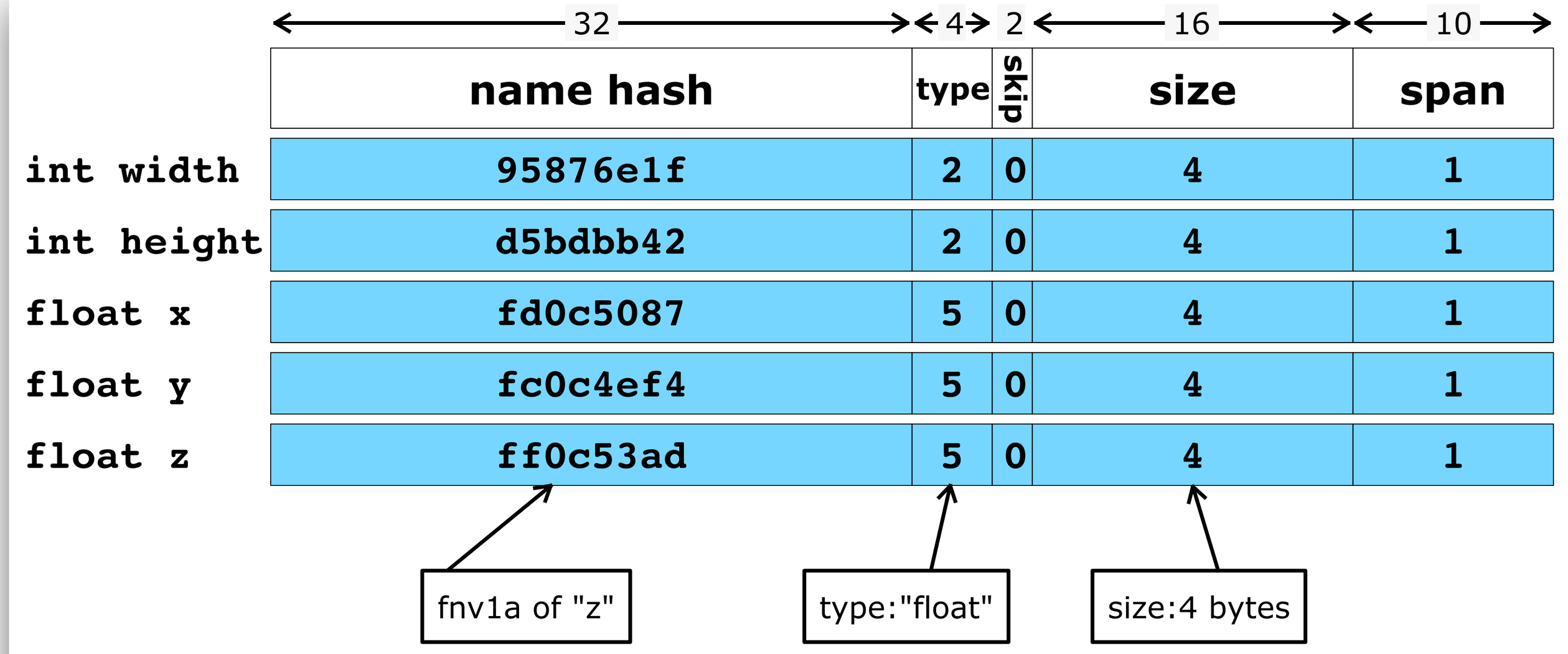


This is a tokenized, compact binary encoding of the C struct, the schema, at the time of creation. This is all we need to know to take the data in the file apart.

A minute ago, I promised a format that doesn't require extra work for the programmers. Where does this extra data come from?

A Insomniac, we have been using a Data Definition Language for quite a while. Every C structure that needs to be serialized, is written in DDL instead. This very much like a C struct definition.
Then we have a data compiler that outputs an equivalent C++ header, and a C++ file with with generated code for accessors and serializers.
The same DDL compiler can output this schema.

In the experimental version that I have running currently, the schema has to be hand made. But once we automate that, the engineers need to do nothing extra. They just continue to use DDL to define their data structures.

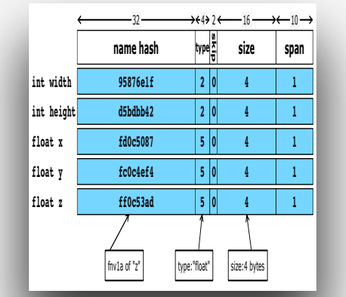


This is a tokenized, compact binary encoding of the C struct, the schema, at the time of creation. This is all we need to know to take the data in the file apart.

A minute ago, I promised a format that doesn't require extra work for the programmers. Where does this extra data come from?

A Insomniac, we have been using a Data Definition Language for quite a while. Every C structure that needs to be serialized, is written in DDL instead. This very much like a C struct definition.
Then we have a data compiler that outputs an equivalent C++ header, and a C++ file with with generated code for accessors and serializers.
The same DDL compiler can output this schema.

In the experimental version that I have running currently, the schema has to be hand made. But once we automate that, the engineers need to do nothing extra. They just continue to use DDL to define their data structures.



The data file is no longer an opaque mystery box.

And note that this is very different from formats like a key/value dictionary. We are not tagging every individual data item. We store just enough information to reconstruct the layout.

Most data files contain many instances of the same type of object. So for example if you were to store a model with a thousand vertices, you would need one schema to describe the model, and one schema to describe a vertex.

If you were to store this in a BSON or Binary XML file, you would have to include X, Y, and Z tags in every vertex. Not with Structured Binary.

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

The data file is no longer an opaque mystery box.

And note that this is very different from formats like a key/value dictionary. We are not tagging every individual data item. We store just enough information to reconstruct the layout.

Most data files contain many instances of the same type of object. So for example if you were to store a model with a thousand vertices, you would need one schema to describe the model, and one schema to describe a vertex.

If you were to store this in a BSON or Binary XML file, you would have to include X, Y, and Z tags in every vertex. Not with Structured Binary.

"Structured Binary" data file

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

en ile

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

runtime loader

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

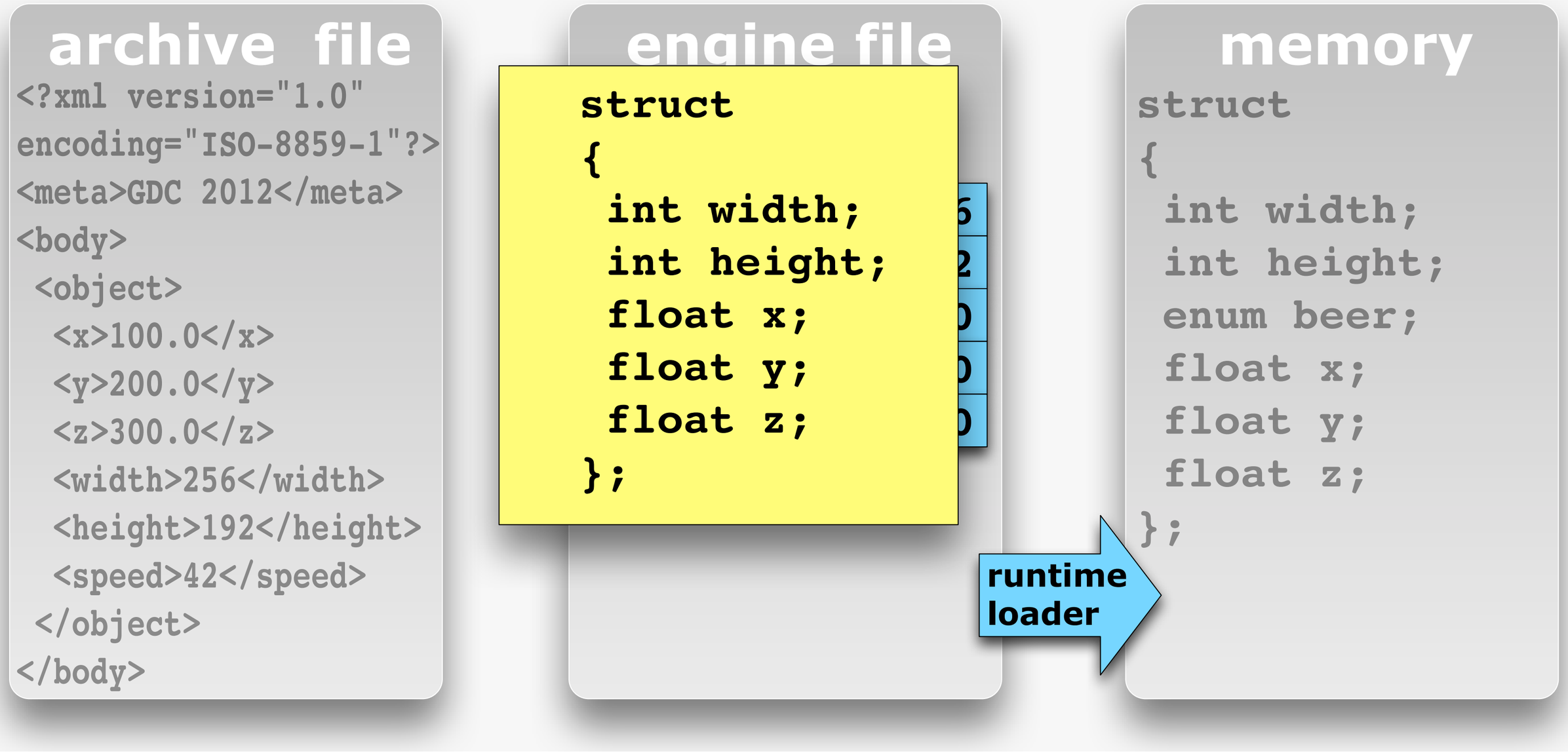
The data file is no longer an opaque mystery box.

And note that this is very different from formats like a key/value dictionary. We are not tagging every individual data item. We store just enough information to reconstruct the layout.

Most data files contain many instances of the same type of object. So for example if you were to store a model with a thousand vertices, you would need one schema to describe the model, and one schema to describe a vertex.

If you were to store this in a BSON or Binary XML file, you would have to include X, Y, and Z tags in every vertex. Not with Structured Binary.

"Structured Binary" data files



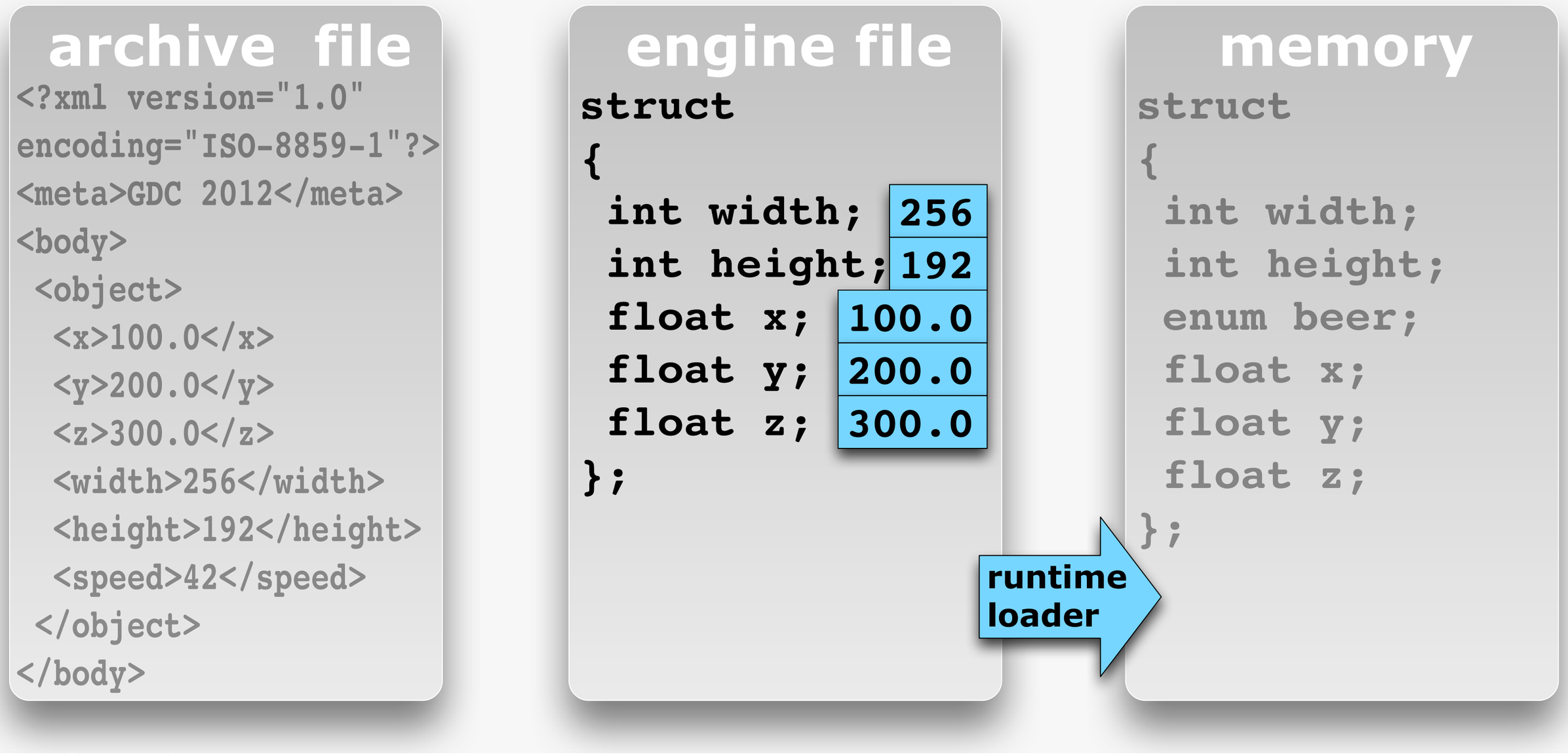
The data file is no longer an opaque mystery box.

And note that this is very different from formats like a key/value dictionary. We are not tagging every individual data item. We store just enough information to reconstruct the layout.

Most data files contain many instances of the same type of object. So for example if you were to store a model with a thousand vertices, you would need one schema to describe the model, and one schema to describe a vertex.

If you were to store this in a BSON or Binary XML file, you would have to include X, Y, and Z tags in every vertex. Not with Structured Binary.

"Structured Binary" data files



The data file is no longer an opaque mystery box.

And note that this is very different from formats like a key/value dictionary. We are not tagging every individual data item. We store just enough information to reconstruct the layout.

Most data files contain many instances of the same type of object. So for example if you were to store a model with a thousand vertices, you would need one schema to describe the model, and one schema to describe a vertex.

If you were to store this in a BSON or Binary XML file, you would have to include X, Y, and Z tags in every vertex. Not with Structured Binary.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

runtime loader

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

Now that we know how the data in the file is laid out, and the runtime knows how the data needs to be laid out in memory, we can match each data field in memory with one in the data file.
[CLICK] This is how we convert the data from the file.
[CLICK] Any missing data is set to a safe default. Any data in the file that is not used in the new memory struct, is ignored. Just as READ-N-BUILD.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

runtime loader

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

Now that we know how the data in the file is laid out, and the runtime knows how the data needs to be laid out in memory, we can match each data field in memory with one in the data file.
[CLICK] This is how we convert the data from the file.
[CLICK] Any missing data is set to a safe default. Any data in the file that is not used in the new memory struct, is ignored. Just as READ-N-BUILD.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

256
192
100.0
200.0
300.0

runtime loader

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

Now that we know how the data in the file is laid out, and the runtime knows how the data needs to be laid out in memory, we can match each data field in memory with one in the data file.
[CLICK] This is how we convert the data from the file.
[CLICK] Any missing data is set to a safe default. Any data in the file that is not used in the new memory struct, is ignored. Just as READ-N-BUILD.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

runtime loader

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

256
192
default
100.0
200.0
300.0

Now that we know how the data in the file is laid out, and the runtime knows how the data needs to be laid out in memory, we can match each data field in memory with one in the data file.
[CLICK] This is how we convert the data from the file.
[CLICK] Any missing data is set to a safe default. Any data in the file that is not used in the new memory struct, is ignored. Just as READ-N-BUILD.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

runtime loader

memory

```
struct
{
 int width;     256
 int height;    192
 enum beer;     default
 float x;       100.0
 float y;       200.0
 float z;       300.0
};
```

Now that we know how the data in the file is laid out, and the runtime knows how the data needs to be laid out in memory, we can match each data field in memory with one in the data file.
[CLICK] This is how we convert the data from the file.
[CLICK] Any missing data is set to a safe default. Any data in the file that is not used in the new memory struct, is ignored. Just as READ-N-BUILD.

"Structured Binary" data files

archive file

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<meta>GDC 2012</meta>
<body>
 <object>
  <x>100.0</x>
  <y>200.0</y>
  <z>300.0</z>
  <width>256</width>
  <height>192</height>
  <speed>42</speed>
 </object>
</body>
```

engine file

```
struct
{
 int width;
 int height;
 float x;
 float y;
 float z;
};
```

memory

```
struct
{
 int width;
 int height;
 enum beer;
 float x;
 float y;
 float z;
};
```

256
192
default
100.0
200.0
300.0

Now that we know how the data in the file is laid out, and the runtime knows how the data needs to be laid out in memory, we can match each data field in memory with one in the data file.
[CLICK] This is how we convert the data from the file.
[CLICK] Any missing data is set to a safe default. Any data in the file that is not used in the new memory struct, is ignored. Just as READ-N-BUILD.

“Structured Binary” data files

Documentation and source code will be available from our website, in a few weeks.

“Structured Binary” data files

- “Load-n-go” when possible

http://www.insomniacgames.com/category/research-development/

Documentation and source code will be available from our website, in a few weeks.

"Structured Binary" data files

- "Load-n-go" when possible
- "Read-n-build" fallback option

http://www.insomniacgames.com/category/research-development/

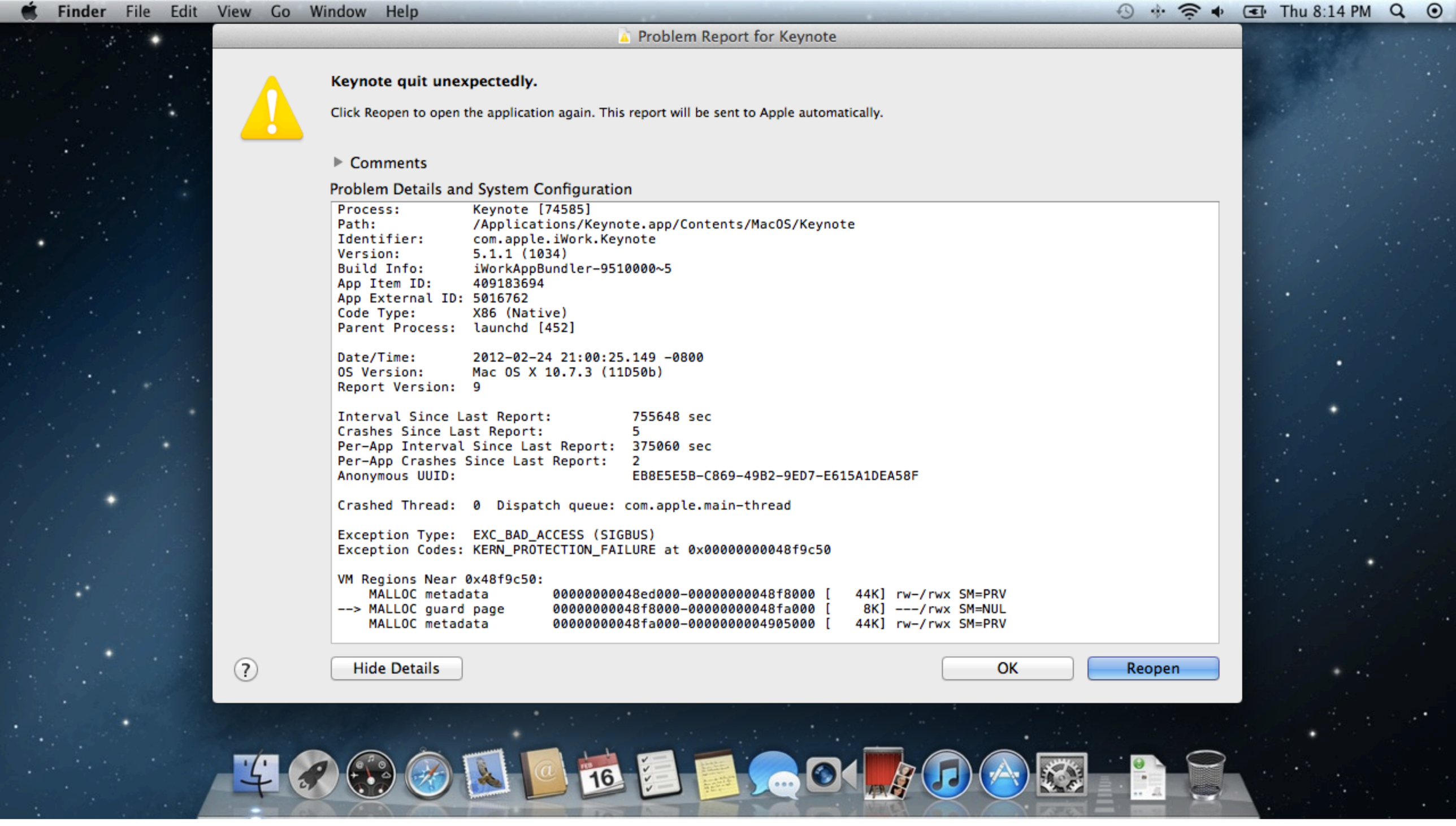
Documentation and source code will be available from our website, in a few weeks.

"Structured Binary" data files

- "Load-n-go" when possible
- "Read-n-build" fallback option
- ...coming soon

http://www.insomniacgames.com/category/research-development/

Documentation and source code will be available from our website, in a few weeks.


Finder File Edit View Go Window Help
Thu 8:14 PM
Problem Report for Keynote
Keynote quit unexpectedly.
Click Reopen to open the application again. This report will be sent to Apple automatically.
Comments
Problem Details and System Configuration
Process: Keynote [74585]
Path: /Applications/Keynote.app/Contents/MacOS/Keynote
Identifier: com.apple.iWork.Keynote
Version: 5.1.1 (1034)
Build Info: iWorkAppBundler-9510000~5
App Item ID: 409183694
App External ID: 5016762
Code Type: X86 (Native)
Parent Process: launchd [452]
Date/Time: 2012-02-24 21:00:25.149 -0800
OS Version: Mac OS X 10.7.3 (11D50b)
Report Version: 9
Interval Since Last Report: 755648 sec
Crashes Since Last Report: 5
Per-App Interval Since Last Report: 375060 sec
Per-App Crashes Since Last Report: 2
Anonymous UUID: EB8E5E5B-C869-49B2-9ED7-E615A1DEA58F
Crashed Thread: 0 Dispatch queue: com.apple.main-thread
Exception Type: EXC_BAD_ACCESS (SIGBUS)
Exception Codes: KERN_PROTECTION_FAILURE at 0x0000000048f9c50
VM Regions Near 0x48f9c50:
MALLOC metadata 00000000048ed000-00000000048f8000 [44K] rw-/rwx SM=PRV
--> MALLOC guard page 00000000048f8000-00000000048fa000 [8K] ---/rwx SM=NUL
MALLOC metadata 00000000048fa000-0000000004905000 [44K] rw-/rwx SM=PRV
?
Hide Details
OK
Reopen

Assertions

I told you I was going to talk about assertions!

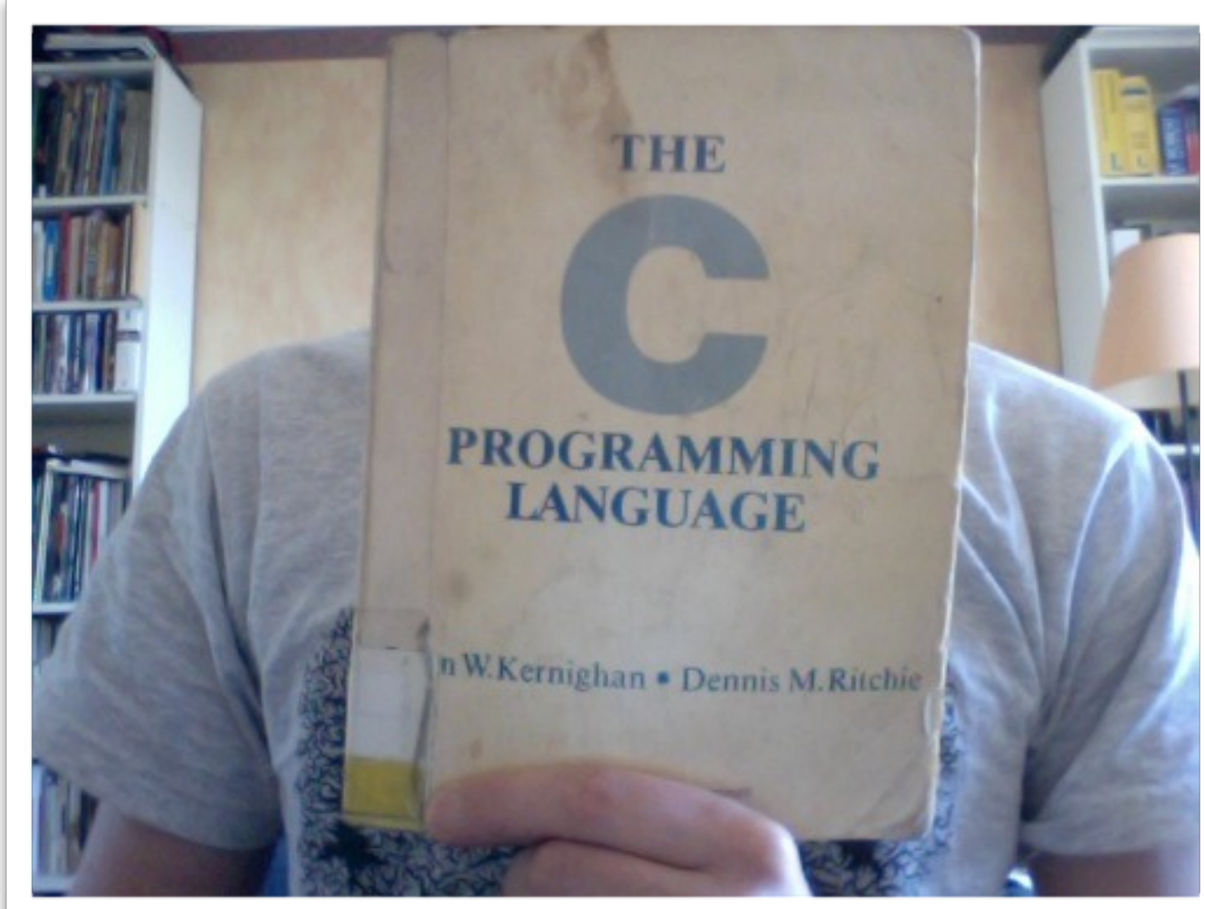
I did a little research, and found that the term “assertion” was coined by, who else, Alan Turing

“In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.”

Turing, A., Checking a Large Routine. in *Conference on High Speed Automatic Calculating Machines*, (Cambridge, UK, 1949), 67-69.

I told you I was going to talk about assertions!

I did a little research, and found that the term “assertion” was coined by, who else, Alan Turing



The C assert macro was introduced by Dennis Ritchie in 1978

The ASSERT Macro

And it has been the programmers best friend for debugging, verifying and testing for three decades, but...

[CLICK] Not the best friend for "using"

Your artists are users, not testers. The assert macro was always intended to be removed in the user build. Unfortunately I don;t see how we can do that.

The ASSERT Macro

- Designed to aid code verification and testing

And it has been the programmers best friend for debugging, verifying and testing for three decades, but...

[CLICK] Not the best friend for "using"

Your artists are users, not testers. The assert macro was always intended to be removed in the user build. Unfortunately I don;t see how we can do that.

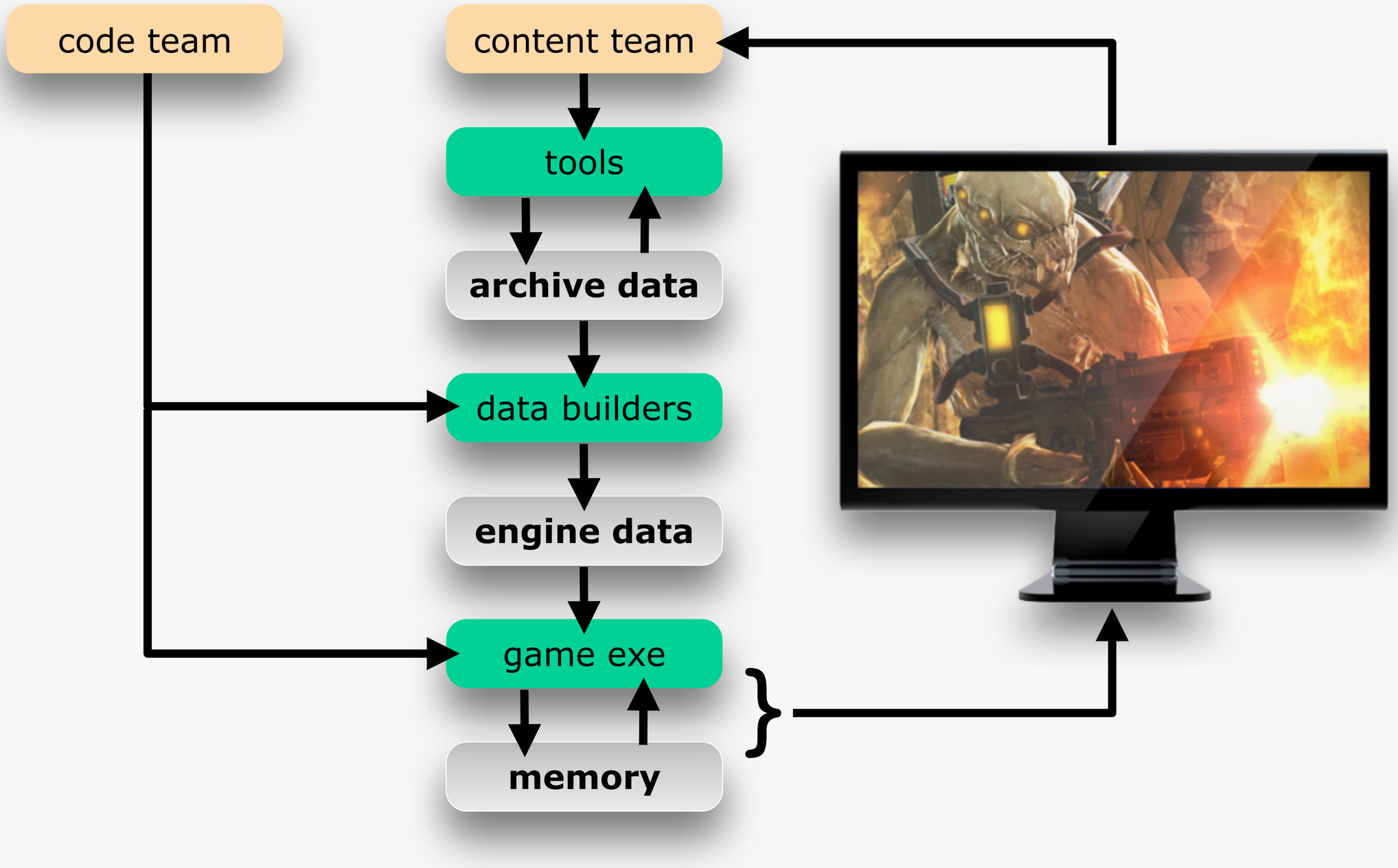
The ASSERT Macro

- Designed to aid code verification and testing
- But our artists are not testing the software, they are using it!

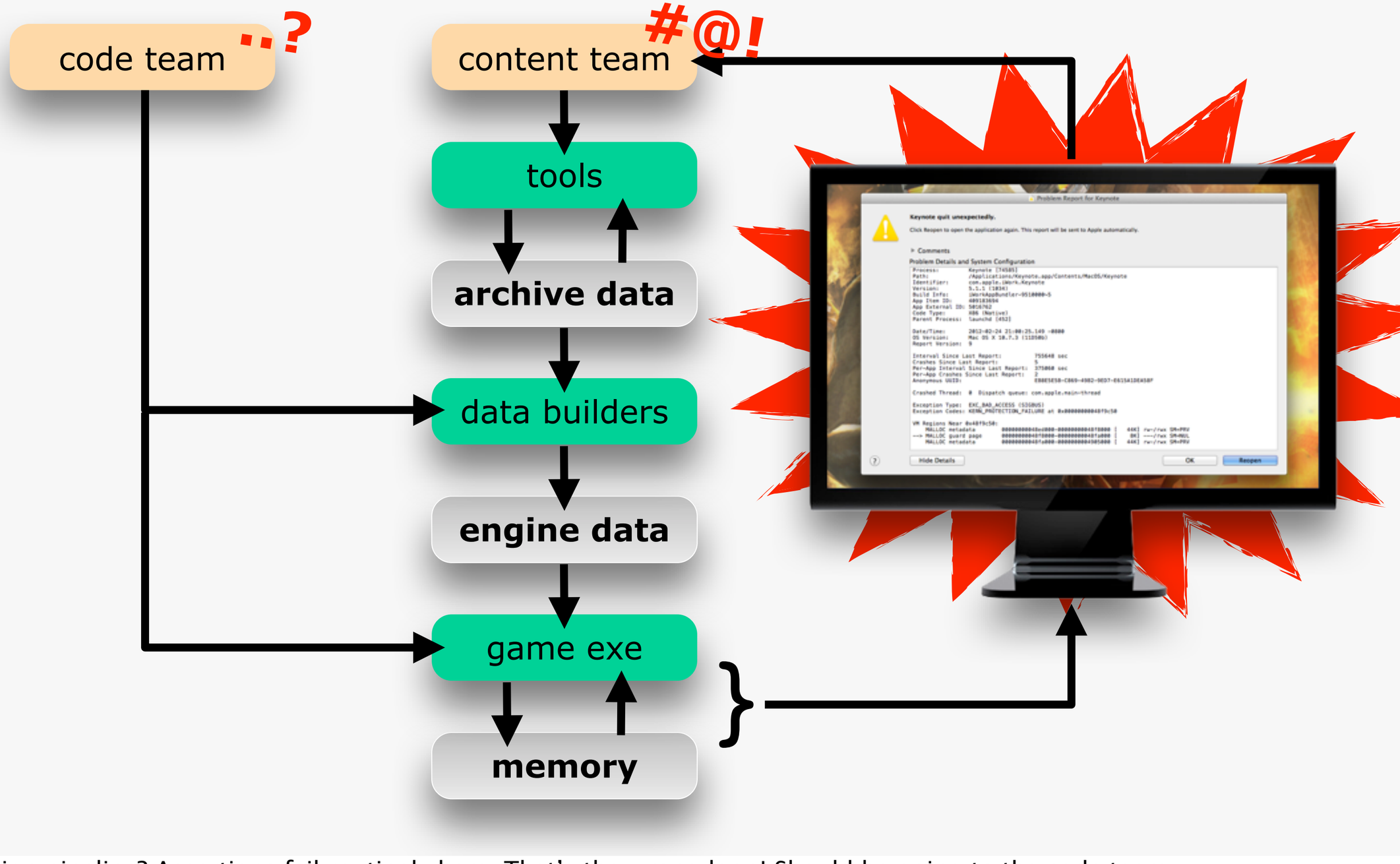
And it has been the programmers best friend for debugging, verifying and testing for three decades, but...

[CLICK] Not the best friend for "using"

Your artists are users, not testers. The assert macro was always intended to be removed in the user build. Unfortunately I don;t see how we can do that.



Remember this production pipeline? Assertions fail routinely here. That’s the wrong loop! Should be going to the code team.



Remember this production pipeline? Assertions fail routinely here. That’s the wrong loop! Should be going to the code team.

Problems with traditional assertion

So the problem is that assertions are commonly presented to the users, your content team.
[CLICK] But these messages are largely irrelevant to the them. They just get in the way. And it is not fair to cast your content team in the role as “the man who checks”. It really isn’t their job.
[CLICK] At the same time, engineers don’t get to see many errors, because they don't use the game in the all the different ways that the content team does. And you can’t reasonably expect your content team to report every assertion box they run in to. They are not “the man who checks”
[CLICK] So “skippable assertions” were invented, and the ignoring began

A skipped assertion is a useless assertion. Why is it even there?

“They never get fixed!” - heard that many times. So we have opposite forces. We want to make assertions more insistent, so people pay attention to them and get forced to fix them. But we also want to make them less obtrusive.

[CLICK] I think we’re trying to solve the wrong problem. The real problem is that traditional assertions are misdirected.

Problems with traditional assertion

- Assertion failures interrupt the workflow of the users

So the problem is that assertions are commonly presented to the users, your content team.
[CLICK] But these messages are largely irrelevant to the them. They just get in the way. And it is not fair to cast your content team in the role as “the man who checks”. It really isn’t their job.
[CLICK] At the same time, engineers don’t get to see many errors, because they don't use the game in the all the different ways that the content team does. And you can’t reasonably expect your content team to report every assertion box they run in to. They are not “the man who checks”
[CLICK] So “skippable assertions” were invented, and the ignoring began

A skipped assertion is a useless assertion. Why is it even there?

“They never get fixed!” - heard that many times. So we have opposite forces. We want to make assertions more insistent, so people pay attention to them and get forced to fix them. But we also want to make them less obtrusive.

[CLICK] I think we’re trying to solve the wrong problem. The real problem is that traditional assertions are misdirected.

Problems with traditional assertion

- Assertion failures interrupt the workflow of the users
- Are meaningless to users, can’t do anything about them

So the problem is that assertions are commonly presented to the users, your content team.
[CLICK] But these messages are largely irrelevant to the them. They just get in the way. And it is not fair to cast your content team in the role as “the man who checks”. It really isn’t their job.
[CLICK] At the same time, engineers don’t get to see many errors, because they don't use the game in the all the different ways that the content team does. And you can’t reasonably expect your content team to report every assertion box they run in to. They are not “the man who checks”
[CLICK] So “skippable assertions” were invented, and the ignoring began

A skipped assertion is a useless assertion. Why is it even there?

“They never get fixed!” - heard that many times. So we have opposite forces. We want to make assertions more insistent, so people pay attention to them and get forced to fix them. But we also want to make them less obtrusive.

[CLICK] I think we’re trying to solve the wrong problem. The real problem is that traditional assertions are misdirected.

Problems with traditional assertion

- Assertion failures interrupt the workflow of the users
- Are meaningless to users, can’t do anything about them
- Engineers don’t get to see assertion reports

So the problem is that assertions are commonly presented to the users, your content team.
[CLICK] But these messages are largely irrelevant to the them. They just get in the way. And it is not fair to cast your content team in the role as “the man who checks”. It really isn’t their job.
[CLICK] At the same time, engineers don’t get to see many errors, because they don't use the game in the all the different ways that the content team does. And you can’t reasonably expect your content team to report every assertion box they run in to. They are not “the man who checks”
[CLICK] So “skippable assertions” were invented, and the ignoring began

A skipped assertion is a useless assertion. Why is it even there?

“They never get fixed!” - heard that many times. So we have opposite forces. We want to make assertions more insistent, so people pay attention to them and get forced to fix them. But we also want to make them less obtrusive.

[CLICK] I think we’re trying to solve the wrong problem. The real problem is that traditional assertions are misdirected.

Problems with traditional assertion

- Assertion failures interrupt the workflow of the users
- Are meaningless to users, can’t do anything about them
- Engineers don’t get to see assertion reports
- “Skippable assertions” end up getting ignored

So the problem is that assertions are commonly presented to the users, your content team.
[CLICK] But these messages are largely irrelevant to the them. They just get in the way. And it is not fair to cast your content team in the role as “the man who checks”. It really isn’t their job.
[CLICK] At the same time, engineers don’t get to see many errors, because they don't use the game in the all the different ways that the content team does. And you can’t reasonably expect your content team to report every assertion box they run in to. They are not “the man who checks”
[CLICK] So “skippable assertions” were invented, and the ignoring began

A skipped assertion is a useless assertion. Why is it even there?

“They never get fixed!” - heard that many times. So we have opposite forces. We want to make assertions more insistent, so people pay attention to them and get forced to fix them. But we also want to make them less obtrusive.

[CLICK] I think we’re trying to solve the wrong problem. The real problem is that traditional assertions are misdirected.

Problems with traditional assertion

- Assertion failures interrupt the workflow of the users
- Are meaningless to users, can’t do anything about them
- Engineers don’t get to see assertion reports
- “Skippable assertions” end up getting ignored
- Directed at the wrong audience!

So the problem is that assertions are commonly presented to the users, your content team.
[CLICK] But these messages are largely irrelevant to the them. They just get in the way. And it is not fair to cast your content team in the role as “the man who checks”. It really isn’t their job.
[CLICK] At the same time, engineers don’t get to see many errors, because they don't use the game in the all the different ways that the content team does. And you can’t reasonably expect your content team to report every assertion box they run in to. They are not “the man who checks”
[CLICK] So “skippable assertions” were invented, and the ignoring began

A skipped assertion is a useless assertion. Why is it even there?

“They never get fixed!” - heard that many times. So we have opposite forces. We want to make assertions more insistent, so people pay attention to them and get forced to fix them. But we also want to make them less obtrusive.

[CLICK] I think we’re trying to solve the wrong problem. The real problem is that traditional assertions are misdirected.

Goal of directed assertion

So what we WANT to do is to reduce the intrusion into the user workflow.
SOME assertions need to go to the user, but the majority need to be directed at an engineer. “The man who checks”
And its those assertions that this section of the talk is about.

[CLICK] And in order to reduce the time it takes to fix a failure, we must not only route the assertion to the right person, but also supply a ton of data for him to work with.

Goal of directed assertion

- Reduce intrusion into user workflow

So what we WANT to do is to reduce the intrusion into the user workflow.
SOME assertions need to go to the user, but the majority need to be directed at an engineer. “The man who checks”
And its those assertions that this section of the talk is about.

[CLICK] And in order to reduce the time it takes to fix a failure, we must not only route the assertion to the right person, but also supply a ton of data for him to work with.

Goal of directed assertion

- Reduce intrusion into user workflow
- Reduce time required to fix the error

So what we WANT to do is to reduce the intrusion into the user workflow.
SOME assertions need to go to the user, but the majority need to be directed at an engineer. “The man who checks”
And its those assertions that this section of the talk is about.

[CLICK] And in order to reduce the time it takes to fix a failure, we must not only route the assertion to the right person, but also supply a ton of data for him to work with.

The assert system needs to...

I started doing this kind of thing in the Mercenaries tool chain. The various data builders were rather crash prone to the point of being almost unusable. When I was set the task of making the tool chain usable again, the first thing I did was make all assertions write a file to a shared folder on a server. On several occasions an artist would approach me, and I could tell him that I knew what he came over for, and that I was already working on the problem. The system I am describing today is a little more sophisticated.
[CLICK] I want the assertion system to be as quiet as possible. In practice, you need to let the user know that something is going awry, so it can't always be completely quiet.
[CLICK] And when I need to fix a problem without direct access to the crashed machine, if I have to debug a problem from just a report, I want that report to contain a lot of detail. A call stack is a good start, but there is a ton more evidence that can be collected.
[CLICK] And I don't want to have to interview the user, the artist. How many times have you had to ask: do you have latest? What were you doing? What level? Did you skip any earlier errors? When did you first run into this? Has anyone else run into it? All this information can be collected automatically, and forwarded to "the man who checks"

(BTW: the following is only partly implemented - the rest is ambition)

The assert system needs to...

- Accurately identify the owner, and direct the report

I started doing this kind of thing in the Mercenaries tool chain. The various data builders were rather crash prone to the point of being almost unusable. When I was set the task of making the tool chain usable again, the first thing I did was make all assertions write a file to a shared folder on a server. On several occasions an artist would approach me, and I could tell him that I knew what he came over for, and that I was already working on the problem. The system I am describing today is a little more sophisticated.
[CLICK] I want the assertion system to be as quiet as possible. In practice, you need to let the user know that something is going awry, so it can't always be completely quiet.
[CLICK] And when I need to fix a problem without direct access to the crashed machine, if I have to debug a problem from just a report, I want that report to contain a lot of detail. A call stack is a good start, but there is a ton more evidence that can be collected.
[CLICK] And I don't want to have to interview the user, the artist. How many times have you had to ask: do you have latest? What were you doing? What level? Did you skip any earlier errors? When did you first run into this? Has anyone else run into it? All this information can be collected automatically, and forwarded to "the man who checks"

(BTW: the following is only partly implemented - the rest is ambition)

The assert system needs to...

- Accurately identify the owner, and direct the report
- Be mostly transparent to others

I started doing this kind of thing in the Mercenaries tool chain. The various data builders were rather crash prone to the point of being almost unusable. When I was set the task of making the tool chain usable again, the first thing I did was make all assertions write a file to a shared folder on a server. On several occasions an artist would approach me, and I could tell him that I knew what he came over for, and that I was already working on the problem. The system I am describing today is a little more sophisticated.
[CLICK] I want the assertion system to be as quiet as possible. In practice, you need to let the user know that something is going awry, so it can't always be completely quiet.
[CLICK] And when I need to fix a problem without direct access to the crashed machine, if I have to debug a problem from just a report, I want that report to contain a lot of detail. A call stack is a good start, but there is a ton more evidence that can be collected.
[CLICK] And I don't want to have to interview the user, the artist. How many times have you had to ask: do you have latest? What were you doing? What level? Did you skip any earlier errors? When did you first run into this? Has anyone else run into it? All this information can be collected automatically, and forwarded to "the man who checks"

(BTW: the following is only partly implemented - the rest is ambition)

The assert system needs to...

- Accurately identify the owner, and direct the report
- Be mostly transparent to others
- Collect detailed state information from the crash

I started doing this kind of thing in the Mercenaries tool chain. The various data builders were rather crash prone to the point of being almost unusable. When I was set the task of making the tool chain usable again, the first thing I did was make all assertions write a file to a shared folder on a server. On several occasions an artist would approach me, and I could tell him that I knew what he came over for, and that I was already working on the problem. The system I am describing today is a little more sophisticated.
[CLICK] I want the assertion system to be as quiet as possible. In practice, you need to let the user know that something is going awry, so it can't always be completely quiet.
[CLICK] And when I need to fix a problem without direct access to the crashed machine, if I have to debug a problem from just a report, I want that report to contain a lot of detail. A call stack is a good start, but there is a ton more evidence that can be collected.
[CLICK] And I don't want to have to interview the user, the artist. How many times have you had to ask: do you have latest? What were you doing? What level? Did you skip any earlier errors? When did you first run into this? Has anyone else run into it? All this information can be collected automatically, and forwarded to "the man who checks"

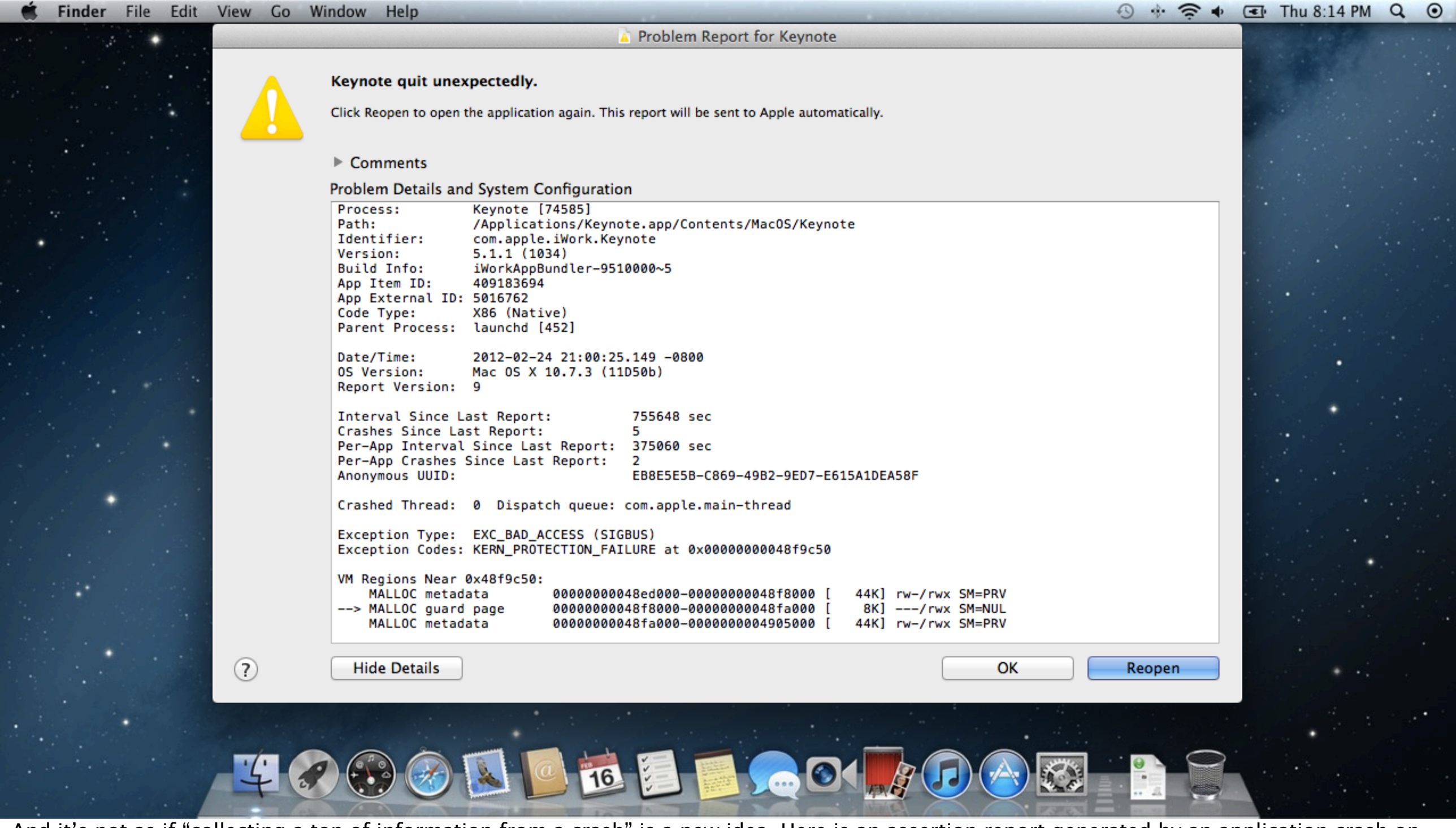
(BTW: the following is only partly implemented - the rest is ambition)

The assert system needs to...

- Accurately identify the owner, and direct the report
- Be mostly transparent to others
- Collect detailed state information from the crash
- Collect a log of events leading up to the failure

I started doing this kind of thing in the Mercenaries tool chain. The various data builders were rather crash prone to the point of being almost unusable. When I was set the task of making the tool chain usable again, the first thing I did was make all assertions write a file to a shared folder on a server. On several occasions an artist would approach me, and I could tell him that I knew what he came over for, and that I was already working on the problem. The system I am describing today is a little more sophisticated.
[CLICK] I want the assertion system to be as quiet as possible. In practice, you need to let the user know that something is going awry, so it can't always be completely quiet.
[CLICK] And when I need to fix a problem without direct access to the crashed machine, if I have to debug a problem from just a report, I want that report to contain a lot of detail. A call stack is a good start, but there is a ton more evidence that can be collected.
[CLICK] And I don't want to have to interview the user, the artist. How many times have you had to ask: do you have latest? What were you doing? What level? Did you skip any earlier errors? When did you first run into this? Has anyone else run into it? All this information can be collected automatically, and forwarded to "the man who checks"

(BTW: the following is only partly implemented - the rest is ambition)



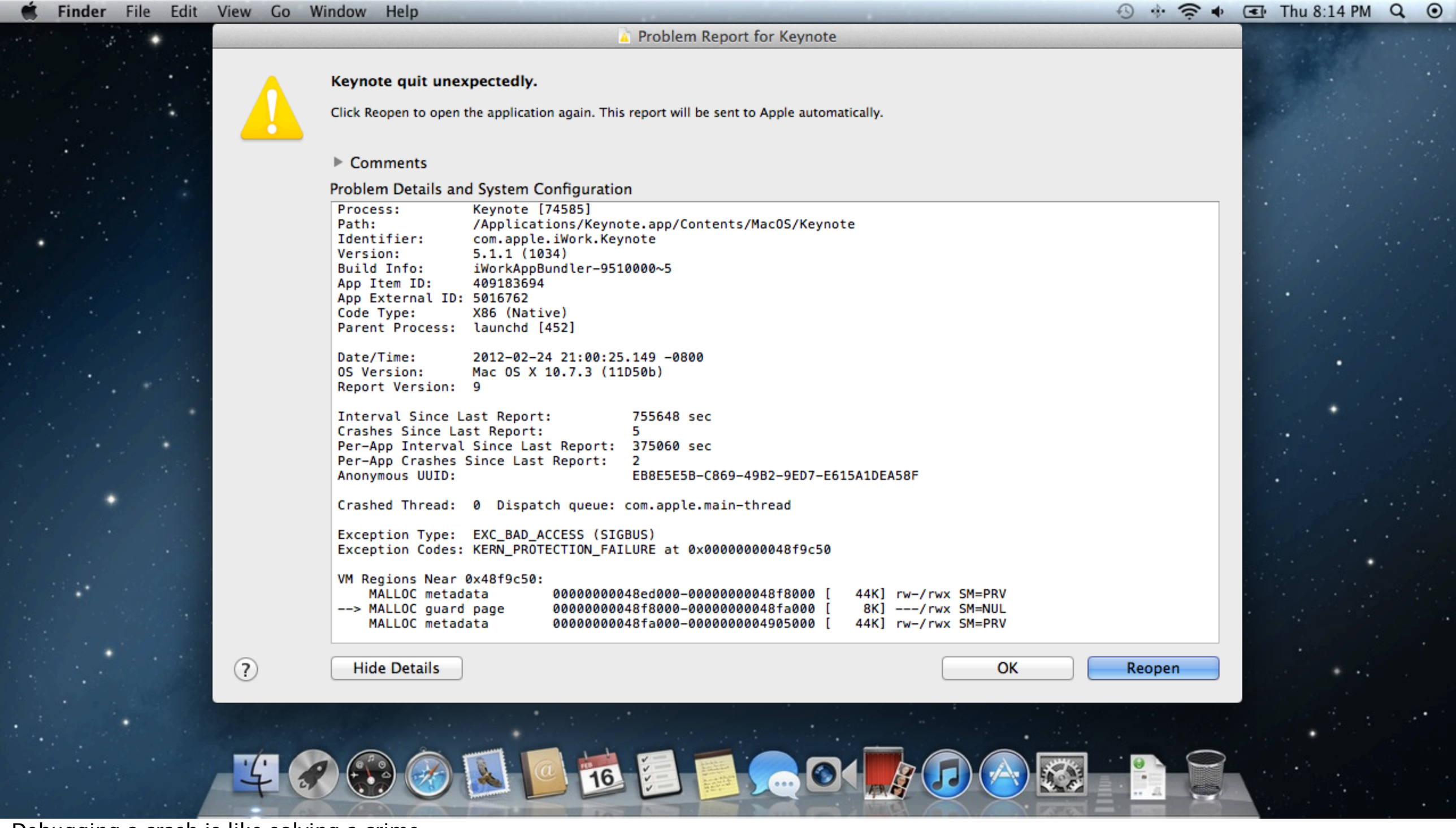
And it's not as if "collecting a ton of information from a crash" is a new idea. Here is an assertion report generated by an application crash on OS X. And when I say "detailed state information" I'm thinking of something like this.

```
Process:         Keynote [74425]
Path:            /Applications/Keynote.app/Contents/MacOS/Keynote
Identifier:      com.apple.iWork.Keynote
Version:         5.1.1 (1034)
Build Info:      iWorkAppBundler-9510000~5
App Item ID:     409183694
App External ID: 5016762
Code Type:       X86 (Native)
Parent Process:  launchd [452]

Date/Time:       2012-02-24 20:54:33.664 -0800
OS Version:      Mac OS X 10.7.3 (11D50b)
Report Version:  9

Crashed Thread:  0  Dispatch queue: com.apple.main-thread

Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Codes: KERN_PROTECTION_FAILURE at 0x00000000048f9c50

VM Regions Near 0x48f9c50:
    MALLOC metadata        00000000048ed000-00000000048f8000 [   44K] rw-/rwx SM=PRV
--> MALLOC guard page      00000000048f8000-00000000048fa000 [    8K] ---/rwx SM=NUL
    MALLOC metadata        00000000048fa000-0000000004905000 [   44K] rw-/rwx SM=PRV

  Application Specific Information:
  objc[74425]: garbage collection is OFF

  Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
  0   libsystem_c.dylib               0x9aee7a40 memmove$VARIANT$sse42 + 131
  1   libGLImage.dylib                0x04618a7a
  _ZL22glgCopyRowsWithMemCopyPK15GLGOperationRecmPK15GLDPixelModeRec + 72
  2   libGLImage.dylib                0x046169f0 glgProcessPixelsWithProcessor + 829
```

And it's not as if "collecting a ton of information from a crash" is a new idea. Here is an assertion report generated by an application crash on OS X. And when I say "detailed state information" I'm thinking of something like this.

And it’s not as if “collecting a ton of information from a crash” is a new idea. Here is an assertion report generated by an application crash on OS X. And when I say “detailed state information” I’m thinking of something like this.

Now THAT’S an assertion report I can work with.



Debugging a crash is like solving a crime.

The police don’t solve a crime AT the crime scene. You can't freeze the crime scene until the crime is solved, life must carry on. So the police send in their forensic team to collect evidence, and the crime is solved later in the forensic lab

Remote debugging on a user machine that has crashed, should be your very last option. This is super intrusive. And if we collect enough evidence, you may rarely have to.

[CLICK] Mercenaries would dump a complete map of all allocated memory, as well as a history of the 5000 most recent memory manager events. This was tremendously helpful. It helped up to track down dangling pointers, memory corruption, memory leaks, causes of fragmentation and so on.

Assertion report

Debugging a crash is like solving a crime.

The police don’t solve a crime AT the crime scene. You can't freeze the crime scene until the crime is solved, life must carry on. So the police send in their forensic team to collect evidence, and the crime is solved later in the forensic lab

Remote debugging on a user machine that has crashed, should be your very last option. This is super intrusive. And if we collect enough evidence, you may rarely have to.

[CLICK] Mercenaries would dump a complete map of all allocated memory, as well as a history of the 5000 most recent memory manager events. This was tremendously helpful. It helped up to track down dangling pointers, memory corruption, memory leaks, causes of fragmentation and so on.

Assertion report

- This is the evidence collected at the crime scene

Debugging a crash is like solving a crime.

The police don’t solve a crime AT the crime scene. You can't freeze the crime scene until the crime is solved, life must carry on. So the police send in their forensic team to collect evidence, and the crime is solved later in the forensic lab

Remote debugging on a user machine that has crashed, should be your very last option. This is super intrusive. And if we collect enough evidence, you may rarely have to.

[CLICK] Mercenaries would dump a complete map of all allocated memory, as well as a history of the 5000 most recent memory manager events. This was tremendously helpful. It helped up to track down dangling pointers, memory corruption, memory leaks, causes of fragmentation and so on.

Assertion report

- This is the evidence collected at the crime scene
- Call stack and registers are only the beginning, also...
- All running processes and threads
- Complete map of all allocated memory blocks
- Time and place of incident
- ...more is better

Debugging a crash is like solving a crime.

The police don’t solve a crime AT the crime scene. You can't freeze the crime scene until the crime is solved, life must carry on. So the police send in their forensic team to collect evidence, and the crime is solved later in the forensic lab

Remote debugging on a user machine that has crashed, should be your very last option. This is super intrusive. And if we collect enough evidence, you may rarely have to.

[CLICK] Mercenaries would dump a complete map of all allocated memory, as well as a history of the 5000 most recent memory manager events. This was tremendously helpful. It helped up to track down dangling pointers, memory corruption, memory leaks, causes of fragmentation and so on.

Session Log

So if the assertion report is your crime scene evidence, the session log is the surveillance tape. This records the events leading up to the crime, and the crime in progress.
This is not a new idea either. All major operating systems have been recording session logs for decades.

Every time your game code does something significant, write it to the log. Every file opened, every read operation completed. Every movement of the user. At Insomniac we send our session data to a central server, so it is immediately accessible by any engineer.

[CLICK] Store everything that could be of interest. The more the better. The only limits are storage and network traffic. And storage is cheap. To minimize network traffic and dependence on network performance, the session data is sent to the local host PC first. A daemon running on the host PC will collect the session log messages and forward them to the log server asynchronously. The data format is also kept very compact. Although the programmer writes to the log system using a printf-style API, the printf formatting is not expanded on the client side at all. The formatting string itself is transmitted, and the printf-style “varargs” are transmitted in their original binary form. Even the formatting string is not transmitted more than once per session.

Session Log

- This is your “surveillance video” of the crime in progress

So if the assertion report is your crime scene evidence, the session log is the surveillance tape. This records the events leading up to the crime, and the crime in progress.
This is not a new idea either. All major operating systems have been recording session logs for decades.

Every time your game code does something significant, write it to the log. Every file opened, every read operation completed. Every movement of the user. At Insomniac we send our session data to a central server, so it is immediately accessible by any engineer.

[CLICK] Store everything that could be of interest. The more the better. The only limits are storage and network traffic. And storage is cheap. To minimize network traffic and dependence on network performance, the session data is sent to the local host PC first. A daemon running on the host PC will collect the session log messages and forward them to the log server asynchronously. The data format is also kept very compact. Although the programmer writes to the log system using a printf-style API, the printf formatting is not expanded on the client side at all. The formatting string itself is transmitted, and the printf-style “varargs” are transmitted in their original binary form. Even the formatting string is not transmitted more than once per session.

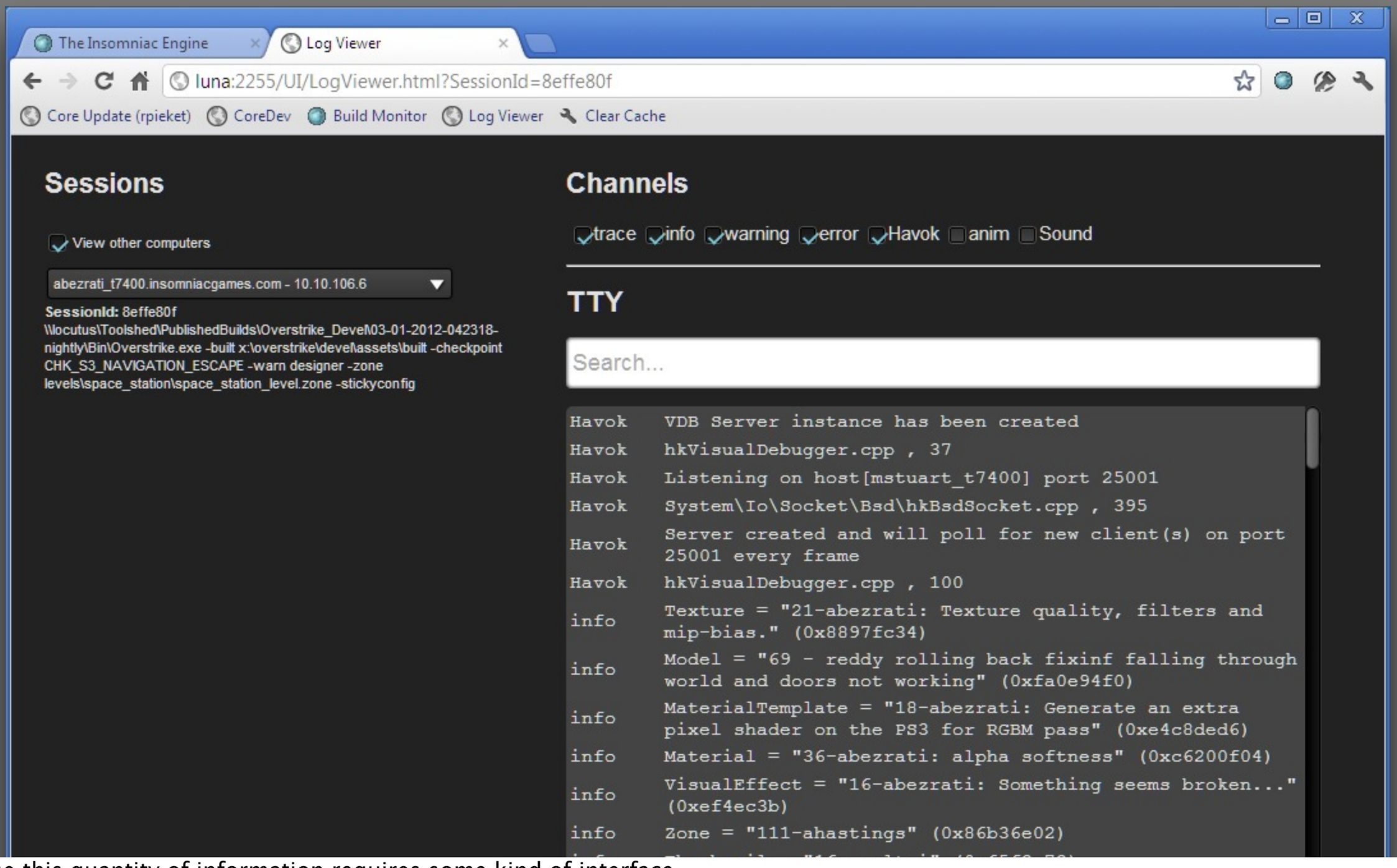
Session Log

- This is your “surveillance video” of the crime in progress
- User name, machine name, code/data versions
- File I/O, asset loading
- Errors, warnings - silent or not
- User actions and state changes
- All skipped assertions in same session

So if the assertion report is your crime scene evidence, the session log is the surveillance tape. This records the events leading up to the crime, and the crime in progress.
This is not a new idea either. All major operating systems have been recording session logs for decades.

Every time your game code does something significant, write it to the log. Every file opened, every read operation completed. Every movement of the user. At Insomniac we send our session data to a central server, so it is immediately accessible by any engineer.

[CLICK] Store everything that could be of interest. The more the better. The only limits are storage and network traffic. And storage is cheap. To minimize network traffic and dependence on network performance, the session data is sent to the local host PC first. A daemon running on the host PC will collect the session log messages and forward them to the log server asynchronously. The data format is also kept very compact. Although the programmer writes to the log system using a printf-style API, the printf formatting is not expanded on the client side at all. The formatting string itself is transmitted, and the printf-style “varargs” are transmitted in their original binary form. Even the formatting string is not transmitted more than once per session.



Of course this quantity of information requires some kind of interface.
At Insomniac, we store our session logs and assertion reports in a MongoDB database with an HTTP server.
This makes it fairly easy to put together front end in a web browser. This screenshot is very much a work in progress.
More sophisticated filtering and searching is in the future. And because the entire database is only an HTTP query away, any programmer can write some javascript to go data mining for something that isn't already covered by the standard interface.

The assert system needs to...

- Accurately identify the owner, and direct the report
- Collect detailed state information from the crash
- Collect a log of events leading up to the failure

Now some code examples.

Remember, we want to make the assertion less intrusive. That means skippable, or even “auto-skip”

```
#define ASSERT( expression, format, ... )\
(\
  ( ( expression ) ?\
    true :\
    (\
      HandleAssert\
      ( __FILE__, __FUNCTION__, __LINE__,\
        #expression, format, __VA_ARGS__\
      ),\
      false\
    )\
  )\
)
```

“Skippable” assertions need a fix-up. The shipped product may, or may not keep the fix up code.

I have slightly restructured the ASSERT macro. You may prefer to give it a different name.

First of all, the ASSERT macro now returns a boolean value. This will control the fix up.

Also, I collect the function name. I will show you why in a minute.

```
#define ASSERT( expression, format, ... )\
(\
  ( ( expression ) ?\
    true :\
    (\
      HandleAssert\
      ( __FILE__, __FUNCTION__, __LINE__,\
        #expression, format, __VA_ARGS__\
      ),\
      false\
    )\
  )\
)
```

“Skippable” assertions need a fix-up. The shipped product may, or may not keep the fix up code.

I have slightly restructured the ASSERT macro. You may prefer to give it a different name.

First of all, the ASSERT macro now returns a boolean value. This will control the fix up.

Also, I collect the function name. I will show you why in a minute.

```
Vec3 VecNormalize( const Vec3& v )
{
  float length = VecLength( v );
  if( ASSERT( length != 0.f ) )
  {
    return v / length;
  }
  else
  {
    return v;
  }
}
```

This is how the new ASSERT macro works with the fix-up code. If the precondition is met, the ASSERT macro returns true, and the normal code can be executed. If the assertion fails, it returns false, and the fix-up occurs.

Of course if the assertion fails, it calls the assertion handler before returning. The assertion handler then reports the assertion to the server, and may or may not display the assertion dialog.

```
Vec3 VecNormalize( const Vec3& v )
{
  float length = VecLength( v );
  if( ASSERT( length != 0.f ) )
  {
    return v / length;
  }
  else
  {
    return v;
  }
}
```

This is how the new ASSERT macro works with the fix-up code. If the precondition is met, the ASSERT macro returns true, and the normal code can be executed. If the assertion fails, it returns false, and the fix-up occurs.

Of course if the assertion fails, it calls the assertion handler before returning. The assertion handler then reports the assertion to the server, and may or may not display the assertion dialog.

```
ScopeOwner* g_ScopeOwner = NULL;

struct ScopeOwner
{
  ScopeOwner( const char* name )
  {
    m_Name = name;
    m_Next = g_ScopeOwner;
    g_ScopeOwner = this;
  }
  ~ScopeOwner() { g_ScopeOwner = m_Next; }

  const char*  m_Name;
  ScopeOwner*  m_Next;
};
```

And if you recall, the system must “accurately identify the owner”.

In previous attempts, we would manually specify the alleged owner as an argument to the macro. This works fine for top-level assertions. What about assertions in the library? Hopefully your libraries are full of assertion tests. Every argument to a library function should be verified. Every operation must be validated. I hope you’re already doing that.

But assertions placed in the library should rarely be directed at the library author. They almost always should go to the author of the code calling in to the library.

That’s why I’m proposing a simple scoping scheme. I’d like to call this feature “blame upcasting”

```
void SomeUserFunction()
{
  ScopeOwner so( "RPW" );
  // ...
  Vec3 dir = VecNormalize( end - start );
  // ...
}
```

This is then how you mark your scope. You create a scoped object on the stack with your ID. The object gets added to the front of the owner list when it is created, and removed when it is destroyed.

Note that you don’t have to declare scope in every single function that you write. You only need to do this in the functions of your system that are part of your API. In other words, your public functions. In a well-designed API, that’s a small number of places.

```
void SomeUserFunction()
{
   ScopeOwner so( "RPW" );
   // ...
   Vec3 dir = VecNormalize( end - start );
   // ...
}
```

This is then how you mark your scope. You create a scoped object on the stack with your ID. The object gets added to the front of the owner list when it is created, and removed when it is destroyed.

Note that you don’t have to declare scope in every single function that you write. You only need to do this in the functions of your system that are part of your API. In other words, your public functions. In a well-designed API, that’s a small number of places.

```
void HandleAssert( const char* file, const char* function,
  int line, const char* expr, const char* fmt, ... )
{
  va_list args;
  va_start( args, fmt );
  const char* owner = g_ScopeOwner->m_Name;
  log::errorf( "Assertion failed in %s, %s, line %d\n",
               file, function, line );
  log::errorf( "Expression %s\n", expr );
  log::errorf( "Id: %s-%s-%s\n", function, file, owner );
  log::errorf( "Owner: %s\n", g_ScopeOwner->m_Name );
  log::verrorf( fmt, args );
  printf( "\n" );
  va_end( args );
}
```

Remember these assertion reports go into a database. And there will be a lot of them. This database needs to be organized in some way. And you definitely want to keep multiple reports of the same assertion together. In order to do that, you need an assertion ID.

Last piece of the puzzle: we need to identify the assert in a persistent manner

The file name and line number won't do for this purpose. You will be collecting reports over a period of time. And if, during that time, the source file changes, the line number of the assertion will change. Identifying the assertion by the function, and not by the line number, is more persistent.

Note that I'm also including the scope owner in the ID. This means that the VecNormalize assertion that is called from, say, the animation system scope will be sorted separate from the VecNormalize called from the AI system scope.

```
void HandleAssert( const char* file, const char* function,
  int line, const char* expr, const char* fmt, ... )
{
  va_list args;
  va_start( args, fmt );
  const char* owner = g_ScopeOwner->m_Name;
  log::errorf( "Assertion failed in %s, %s, line %d\n",
               file, function, line );
  log::errorf( "Expression %s\n", expr );
  log::errorf( "Id: %s-%s-%s\n", function, file, owner );
  log::errorf( "Owner: %s\n", g_ScopeOwner->m_Name );
  log::verrorf( fmt, args );
  printf( "\n" );
  va_end( args );
}
```

Remember these assertion reports go into a database. And there will be a lot of them. This database needs to be organized in some way. And you definitely want to keep multiple reports of the same assertion together. In order to do that, you need an assertion ID.

Last piece of the puzzle: we need to identify the assert in a persistent manner

The file name and line number won't do for this purpose. You will be collecting reports over a period of time. And if, during that time, the source file changes, the line number of the assertion will change. Identifying the assertion by the function, and not by the line number, is more persistent.

Note that I'm also including the scope owner in the ID. This means that the VecNormalize assertion that is called from, say, the animation system scope will be sorted separate from the VecNormalize called from the AI system scope.

```
//DEBUG
  if( ASSERT( length != 0.f ) )
    return v / length;
  else
    return v;
```

Traditionally, or at least, as originally intended by Dennis Ritchie in 1978, assertion macros are not compiled for release builds.
But I think most developers would like to keep at least the fix-up code in the shipped version.
The syntax that I just proposed, with the ASSERT macro that returns a boolean, you have a number of options. You will need to define a few different versions of the ASSERT macro, that compile differently under a RELEASE build.
[CLICK] So if you want to keep the fix up in a release build, you can rig your ASSERT macro in such a way that after the preprocessor, the code looks like this. Only the test and the fix-up are kept.
[CLICK] Some assertion tests may cause noticeable performance issues. This is the case for assertions in inner loops, or assertions that involve expensive verification. In that case, you'd rig a version of the ASSERT macro that expands just to the boolean value TRUE.
[CLICK]The optimizer will strip out the unreachable code.

```
//DEBUG
  if( ASSERT( length != 0.f ) )
    return v / length;
  else
    return v;
```

Traditionally, or at least, as originally intended by Dennis Ritchie in 1978, assertion macros are not compiled for release builds.
But I think most developers would like to keep at least the fix-up code in the shipped version.
The syntax that I just proposed, with the ASSERT macro that returns a boolean, you have a number of options. You will need to define a few different versions of the ASSERT macro, that compile differently under a RELEASE build.
[CLICK] So if you want to keep the fix up in a release build, you can rig your ASSERT macro in such a way that after the preprocessor, the code looks like this. Only the test and the fix-up are kept.
[CLICK] Some assertion tests may cause noticeable performance issues. This is the case for assertions in inner loops, or assertions that involve expensive verification. In that case, you'd rig a version of the ASSERT macro that expands just to the boolean value TRUE.
[CLICK]The optimizer will strip out the unreachable code.

```
//DEBUG
  if( ASSERT( length != 0.f ) )
    return v / length;
  else
    return v;

//RELEASE "better safe than sorry" version
  if( length != 0.f )
    return v / length;
  else
    return v;
```

Traditionally, or at least, as originally intended by Dennis Ritchie in 1978, assertion macros are not compiled for release builds.
But I think most developers would like to keep at least the fix-up code in the shipped version.
The syntax that I just proposed, with the ASSERT macro that returns a boolean, you have a number of options. You will need to define a few different versions of the ASSERT macro, that compile differently under a RELEASE build.
[CLICK] So if you want to keep the fix up in a release build, you can rig your ASSERT macro in such a way that after the preprocessor, the code looks like this. Only the test and the fix-up are kept.
[CLICK] Some assertion tests may cause noticeable performance issues. This is the case for assertions in inner loops, or assertions that involve expensive verification. In that case, you'd rig a version of the ASSERT macro that expands just to the boolean value TRUE.
[CLICK]The optimizer will strip out the unreachable code.

```
//DEBUG
  if( ASSERT( length != 0.f ) )
    return v / length;
  else
    return v;

//RELEASE "better safe than sorry" version
  if( length != 0.f )
    return v / length;
  else
    return v;
```

Traditionally, or at least, as originally intended by Dennis Ritchie in 1978, assertion macros are not compiled for release builds.
But I think most developers would like to keep at least the fix-up code in the shipped version.
The syntax that I just proposed, with the ASSERT macro that returns a boolean, you have a number of options. You will need to define a few different versions of the ASSERT macro, that compile differently under a RELEASE build.
[CLICK] So if you want to keep the fix up in a release build, you can rig your ASSERT macro in such a way that after the preprocessor, the code looks like this. Only the test and the fix-up are kept.
[CLICK] Some assertion tests may cause noticeable performance issues. This is the case for assertions in inner loops, or assertions that involve expensive verification. In that case, you'd rig a version of the ASSERT macro that expands just to the boolean value TRUE.
[CLICK]The optimizer will strip out the unreachable code.

```
//DEBUG
  if( ASSERT( length != 0.f ) )
    return v / length;
  else
    return v;
```

```
//RELEASE "better safe than sorry" version
  if( length != 0.f )
    return v / length;
  else
    return v;
```

```
//RELEASE "need moar speed" version
  if( true )
    return v / length;
  else
    return v;
```

Traditionally, or at least, as originally intended by Dennis Ritchie in 1978, assertion macros are not compiled for release builds.
But I think most developers would like to keep at least the fix-up code in the shipped version.
The syntax that I just proposed, with the ASSERT macro that returns a boolean, you have a number of options. You will need to define a few different versions of the ASSERT macro, that compile differently under a RELEASE build.
[CLICK] So if you want to keep the fix up in a release build, you can rig your ASSERT macro in such a way that after the preprocessor, the code looks like this. Only the test and the fix-up are kept.
[CLICK] Some assertion tests may cause noticeable performance issues. This is the case for assertions in inner loops, or assertions that involve expensive verification. In that case, you'd rig a version of the ASSERT macro that expands just to the boolean value TRUE.
[CLICK]The optimizer will strip out the unreachable code.

```
//DEBUG
  if( ASSERT( length != 0.f ) )
    return v / length;
  else
    return v;

//RELEASE "better safe than sorry" version
  if( length != 0.f )
    return v / length;
  else
    return v;

//RELEASE "need moar speed" version
  if( true )
    return v / length;
  else
    return v;
```

Traditionally, or at least, as originally intended by Dennis Ritchie in 1978, assertion macros are not compiled for release builds.
But I think most developers would like to keep at least the fix-up code in the shipped version.
The syntax that I just proposed, with the ASSERT macro that returns a boolean, you have a number of options. You will need to define a few different versions of the ASSERT macro, that compile differently under a RELEASE build.
[CLICK] So if you want to keep the fix up in a release build, you can rig your ASSERT macro in such a way that after the preprocessor, the code looks like this. Only the test and the fix-up are kept.
[CLICK] Some assertion tests may cause noticeable performance issues. This is the case for assertions in inner loops, or assertions that involve expensive verification. In that case, you'd rig a version of the ASSERT macro that expands just to the boolean value TRUE.
[CLICK]The optimizer will strip out the unreachable code.

Directed assertions

So to summarize: directed assertions use a scoping mechanism to identify the likely owner, and more accurately direct the assertion.
[CLICK] Collect a lot more data from the crime scene than just a call stack
[CLICK] Record all events leading up to the failure
[CLICK] All this is done in an effort to solve the crime without involving innocent bystanders
[CLICK] It is really a remote debugging system. Remote in location, and also in time

Directed assertions

- Report assertions to the right team member

So to summarize: directed assertions use a scoping mechanism to identify the likely owner, and more accurately direct the assertion.
[CLICK] Collect a lot more data from the crime scene than just a call stack
[CLICK] Record all events leading up to the failure
[CLICK] All this is done in an effort to solve the crime without involving innocent bystanders
[CLICK] It is really a remote debugging system. Remote in location, and also in time

Directed assertions

- Report assertions to the right team member
- CSI: collect as much evidence as possible

So to summarize: directed assertions use a scoping mechanism to identify the likely owner, and more accurately direct the assertion.
[CLICK] Collect a lot more data from the crime scene than just a call stack
[CLICK] Record all events leading up to the failure
[CLICK] All this is done in an effort to solve the crime without involving innocent bystanders
[CLICK] It is really a remote debugging system. Remote in location, and also in time

Directed assertions

- Report assertions to the right team member
- CSI: collect as much evidence as possible
- Record extensive session log

So to summarize: directed assertions use a scoping mechanism to identify the likely owner, and more accurately direct the assertion.
[CLICK] Collect a lot more data from the crime scene than just a call stack
[CLICK] Record all events leading up to the failure
[CLICK] All this is done in an effort to solve the crime without involving innocent bystanders
[CLICK] It is really a remote debugging system. Remote in location, and also in time

Directed assertions

- Report assertions to the right team member
- CSI: collect as much evidence as possible
- Record extensive session log
- Don’t bug others about your debugging task

So to summarize: directed assertions use a scoping mechanism to identify the likely owner, and more accurately direct the assertion.
[CLICK] Collect a lot more data from the crime scene than just a call stack
[CLICK] Record all events leading up to the failure
[CLICK] All this is done in an effort to solve the crime without involving innocent bystanders
[CLICK] It is really a remote debugging system. Remote in location, and also in time

Directed assertions

- Report assertions to the right team member
- CSI: collect as much evidence as possible
- Record extensive session log
- Don’t bug others about your debugging task
- More data = quicker fix and less intrusion

So to summarize: directed assertions use a scoping mechanism to identify the likely owner, and more accurately direct the assertion.
[CLICK] Collect a lot more data from the crime scene than just a call stack
[CLICK] Record all events leading up to the failure
[CLICK] All this is done in an effort to solve the crime without involving innocent bystanders
[CLICK] It is really a remote debugging system. Remote in location, and also in time

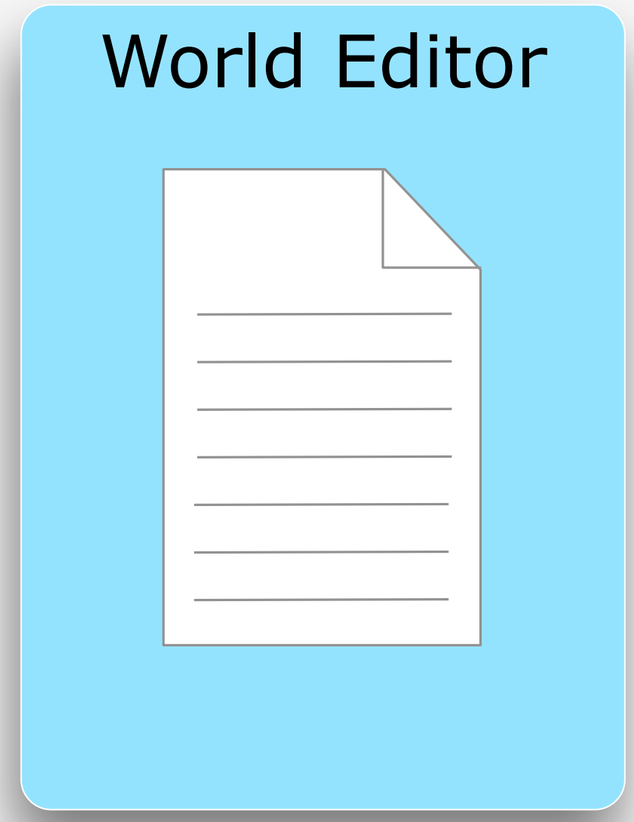
Client/server tools architecture

Now I want to talk about a very powerful architecture choice that we made at Insomniac Games for our tools.

It has been mentioned in other talks already, earlier this week.

This choice was made before I started at Insomniac Games. I have called it the gift that keeps on giving.

Save often

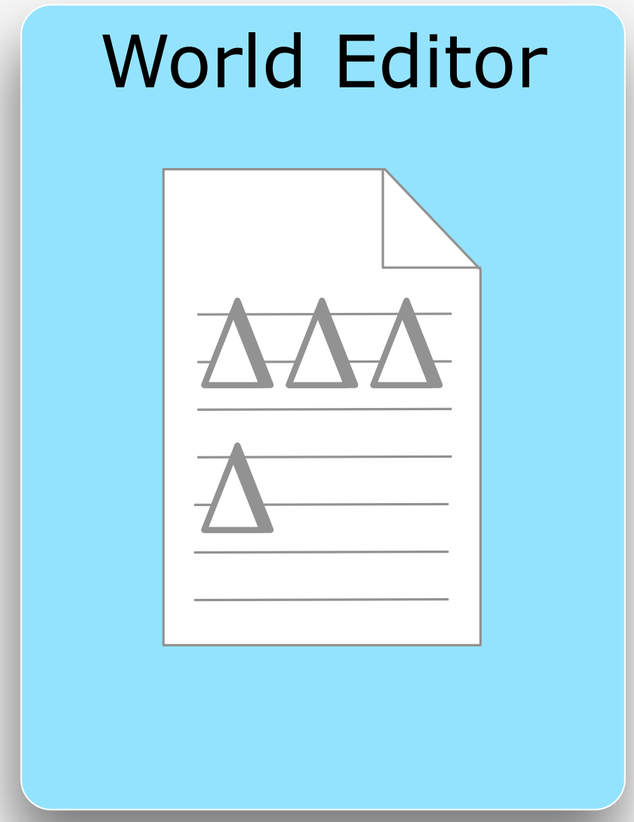


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

Save often

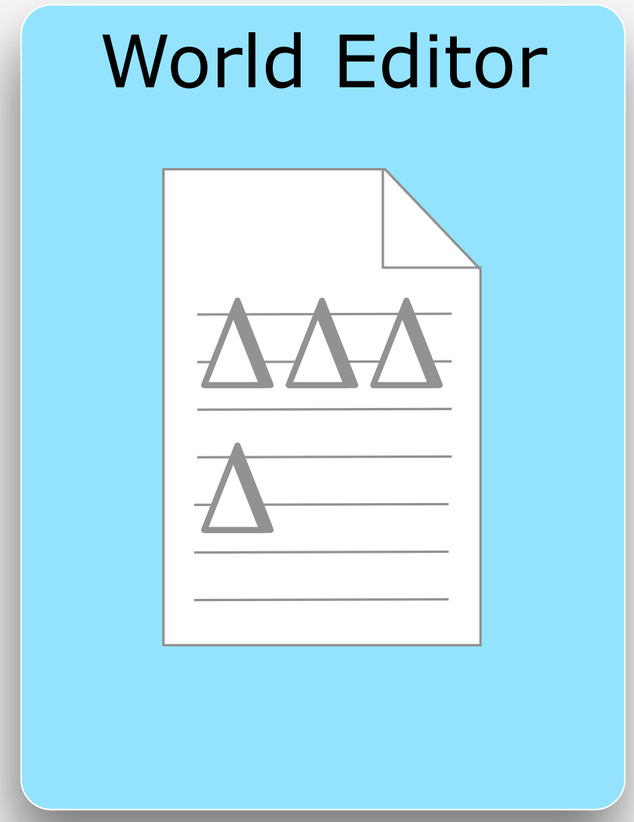


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

Save often

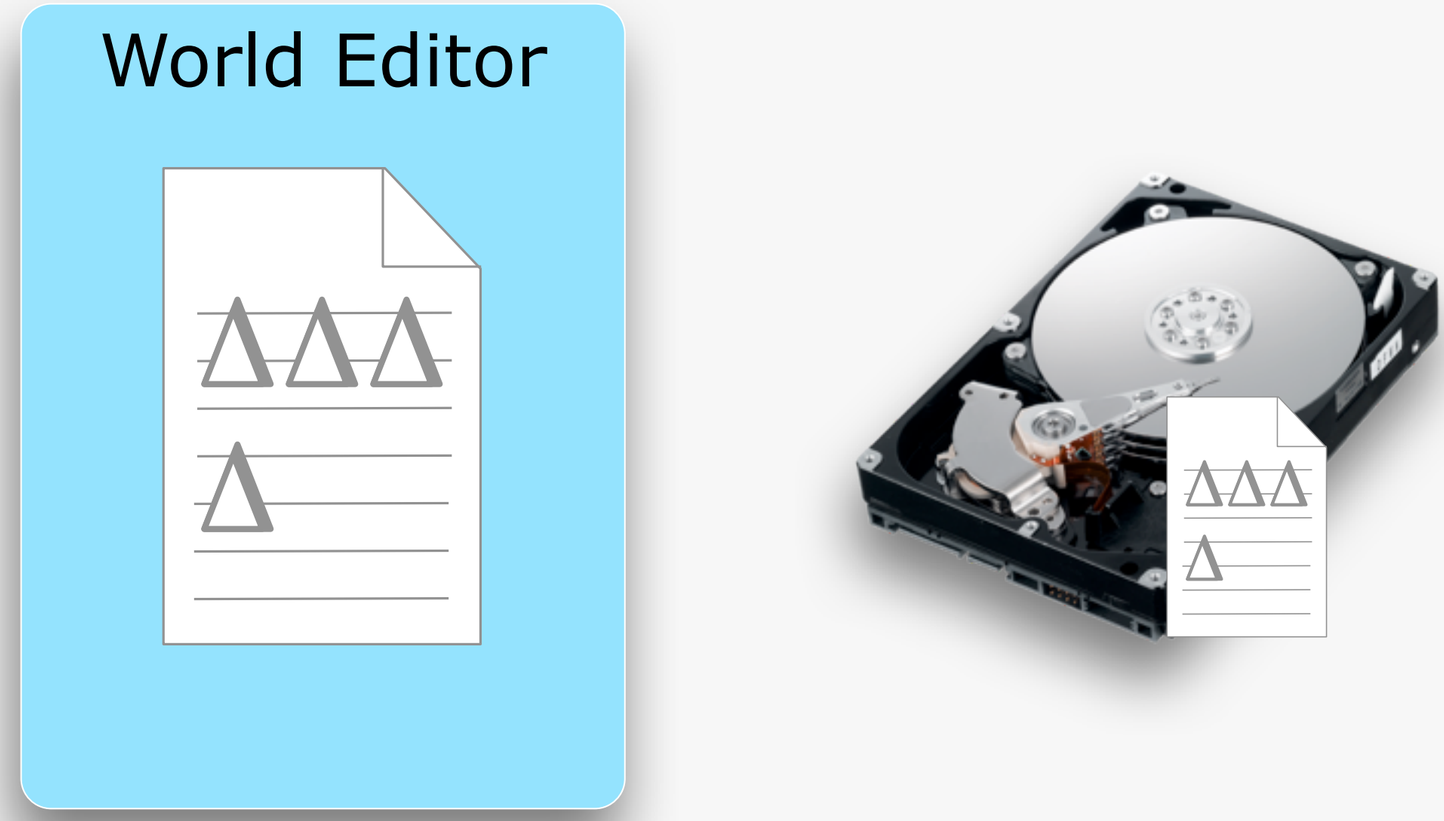


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

Save often

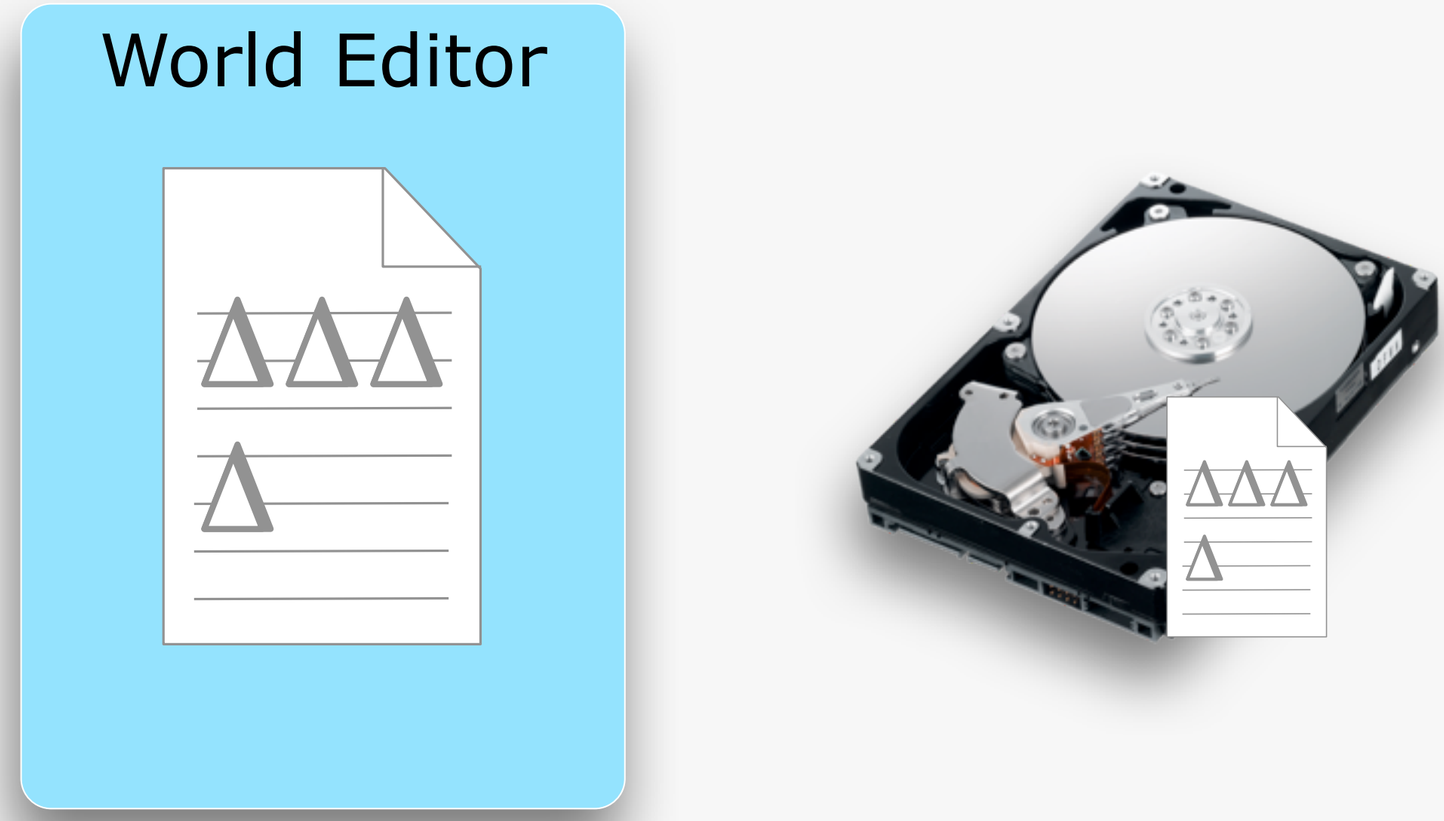


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

Save often

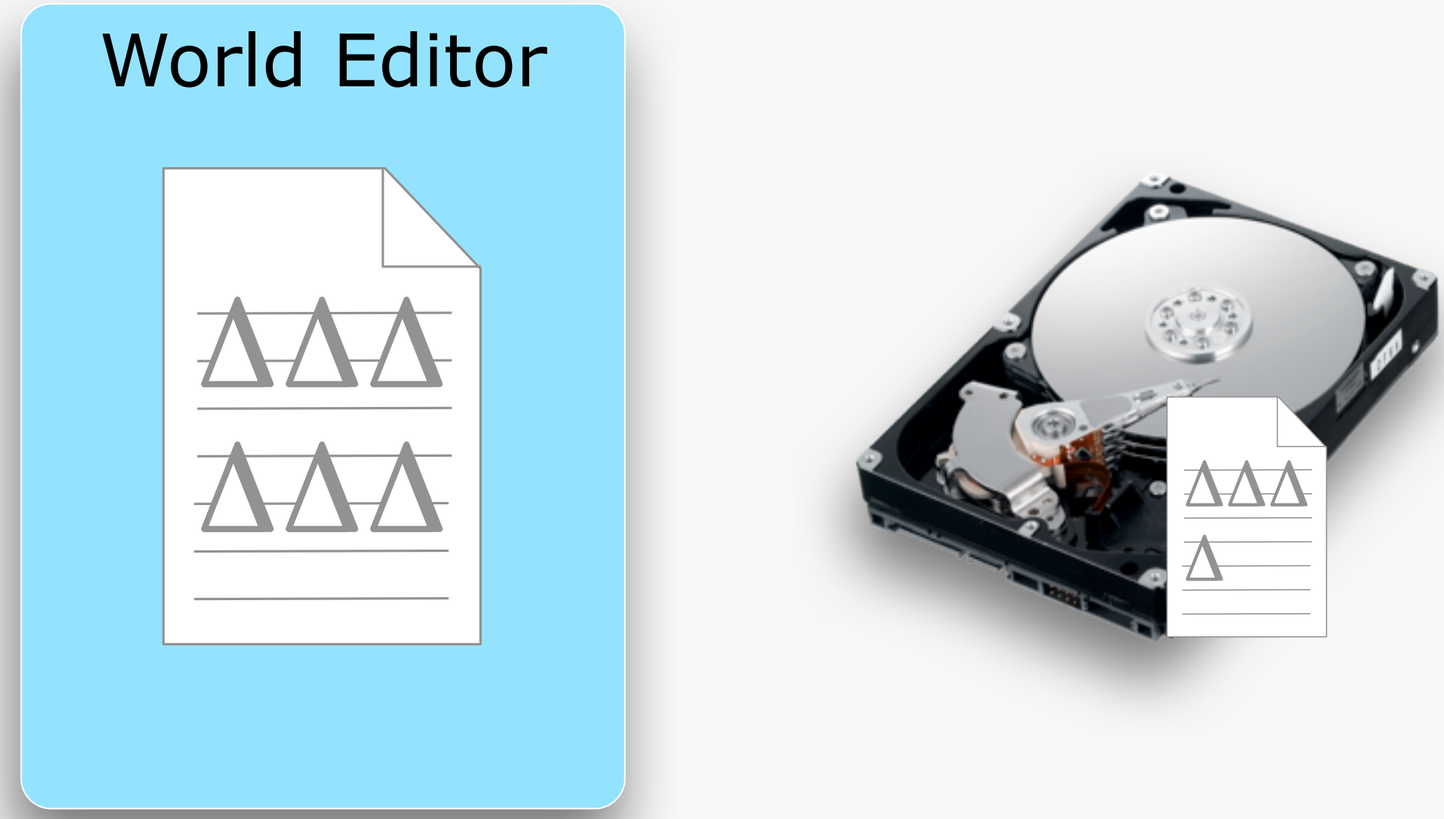


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

Save often

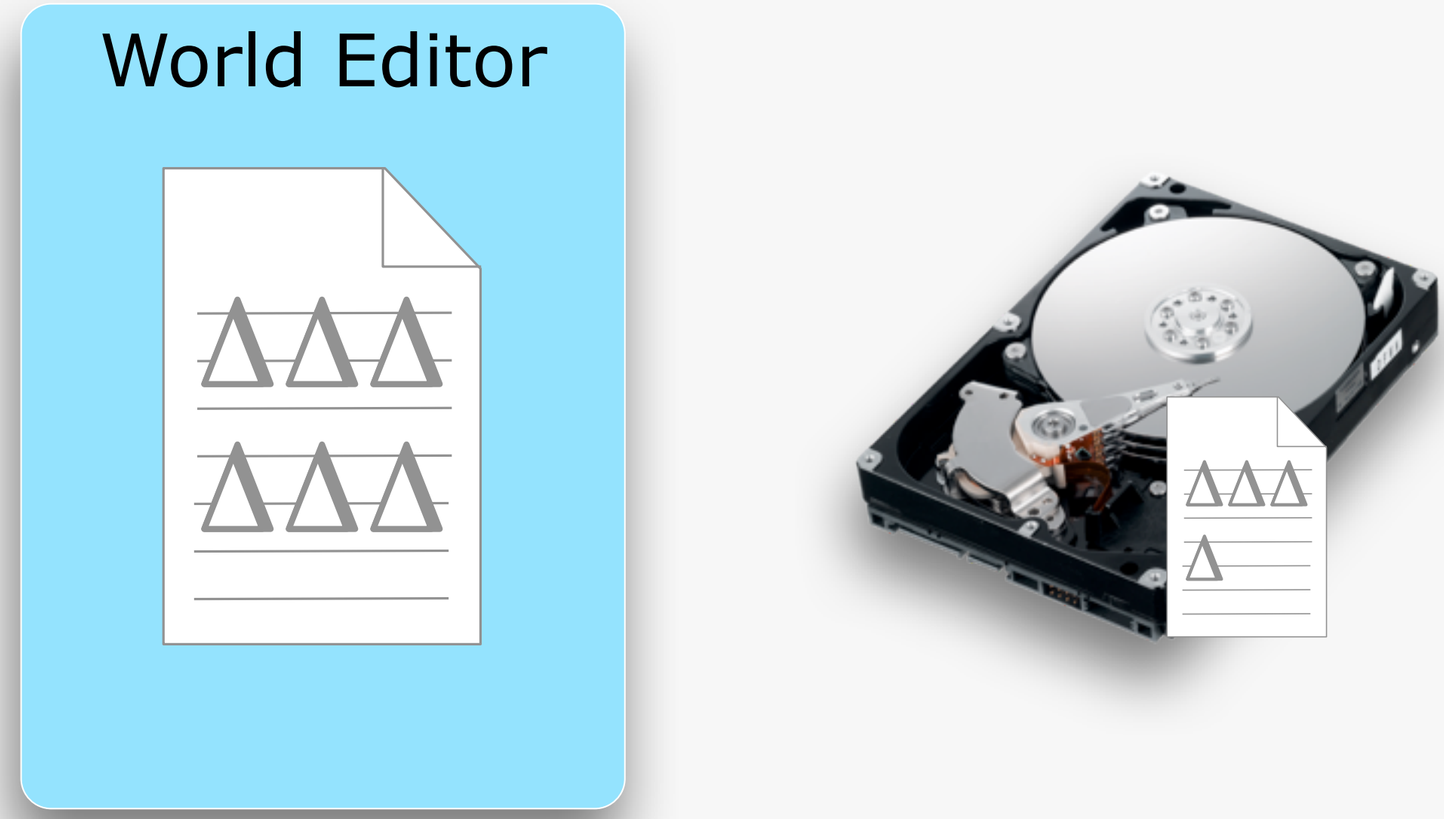


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

Save often

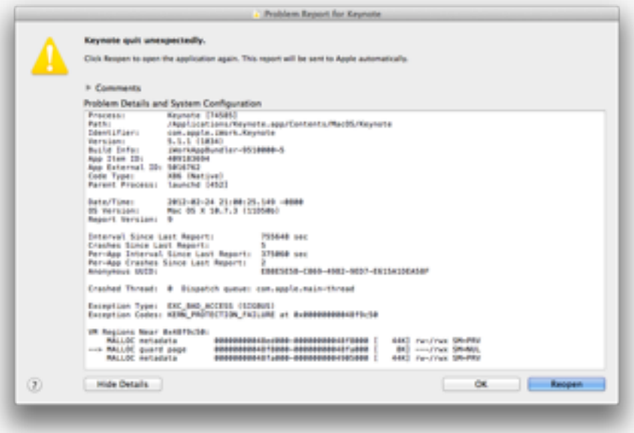


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

Save often

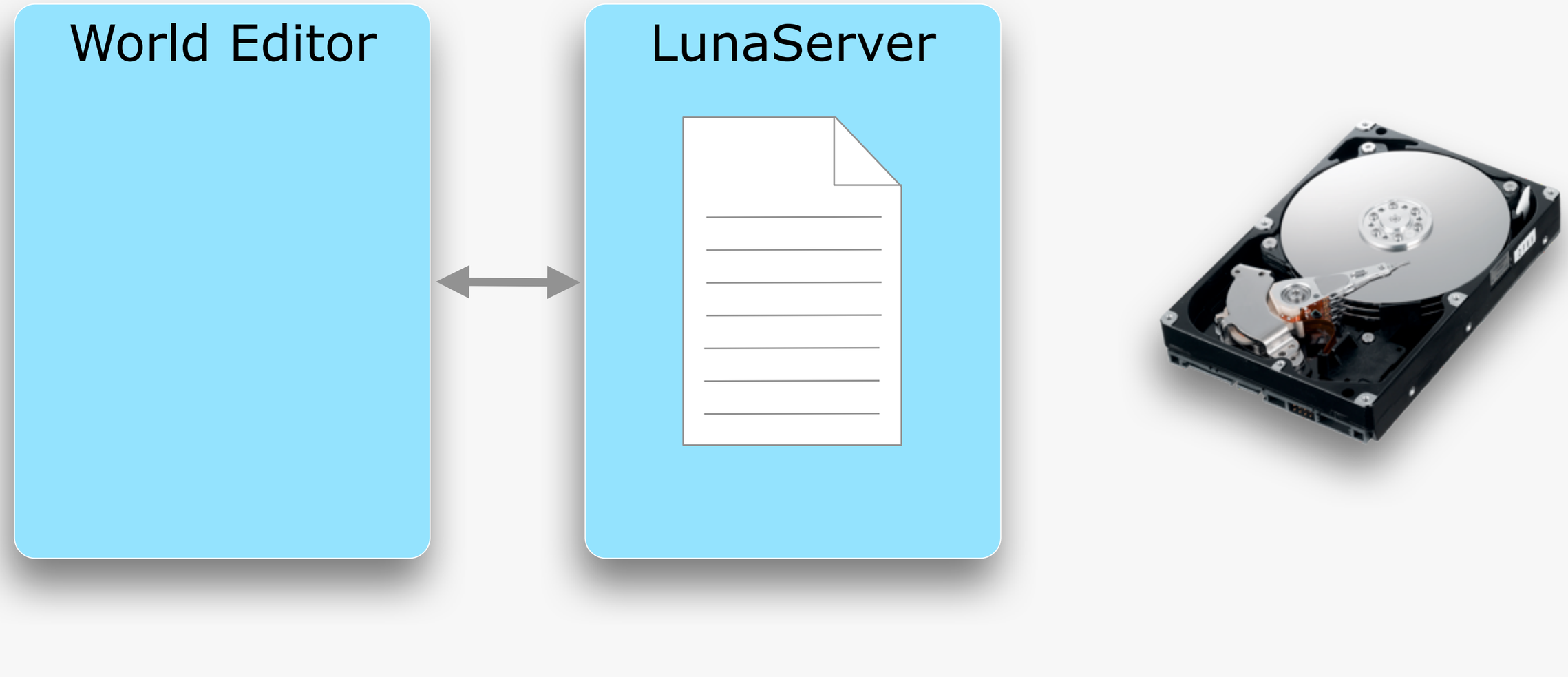


Let me use the example of our world editor.

In a traditional editor, the document that you are working on is kept and maintained by the editor application itself. This is the working data, or the “model” of a model-view-controller architecture. It is deserialized on start up, and serialized when you save.

[CLICK] So the changes that you make operate on the document model in memory.
Now you have been in this line of work for a while, and you know that the tools are still in development, and only stable in between crashes. So you save often.
[CLICK] And you carry on working
[CLICK] And... [CLICK] boom! There it goes again. Crashed. And with the application, your latest changes are also gone. And so is your copy/paste data and undo queue.

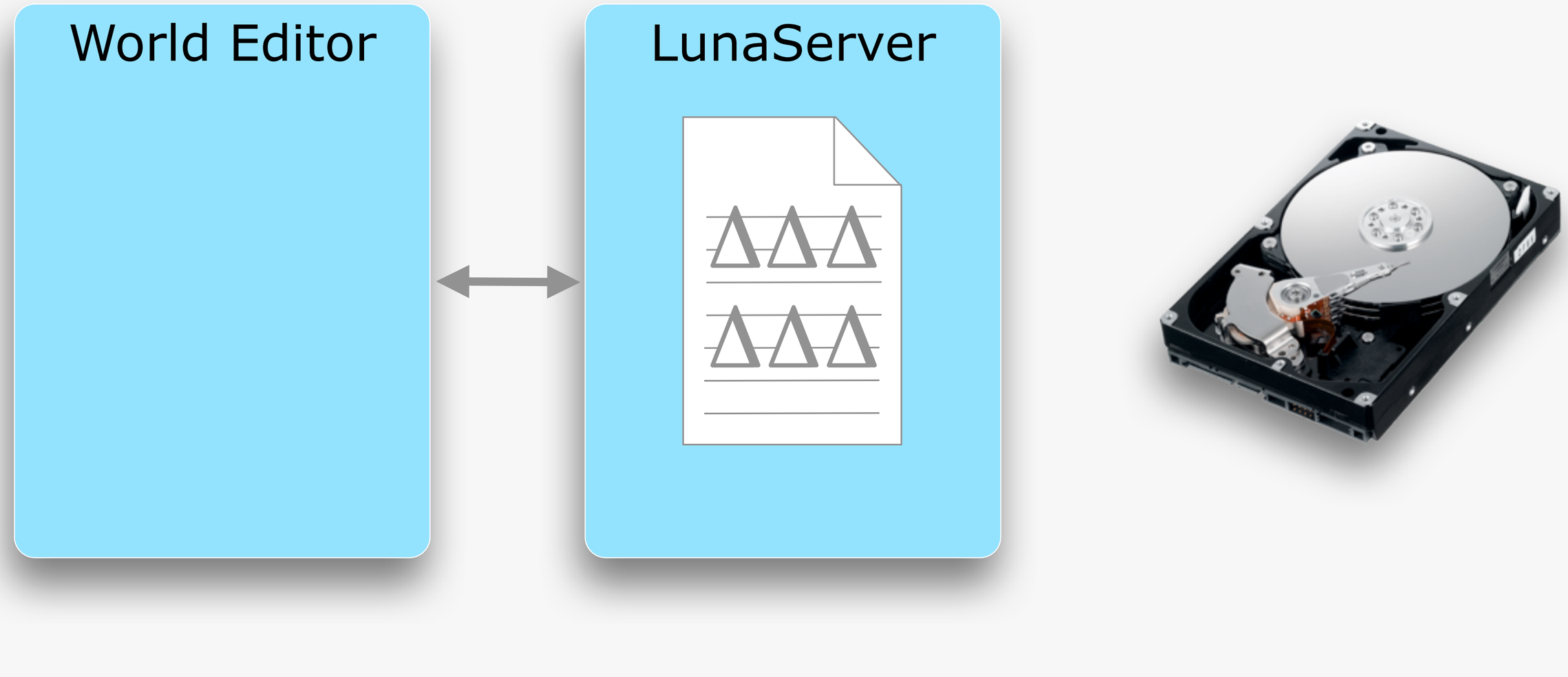
Save never



In the client-server architecture that we developed for our tools, this works quite differently. The editor application doesn't store the document. It is kept on a server. We call it LunaServer.
LunaServer is a process running on the same host PC as the editor. The editor and the server are on the same machine. We have not found it necessary yet to connect over the network.
The application only keeps a cache copy in memory. It does this only so that it can display the document, and process mouse clicks and so on. The authoritative document is maintained by the server.
The server is completely generic. It knows nothing of the nature of the data that it maintains. It just stores the data, collects changes from the client, reports changes to the client when they come in from other clients. More about that in a moment.

[CLICK] While you are using the editor application, it sends every individual change to the server, immediately. The document in its entirety is only ever transmitted once, from the server to the application, when the application starts up. So it can initialize its local cache version.

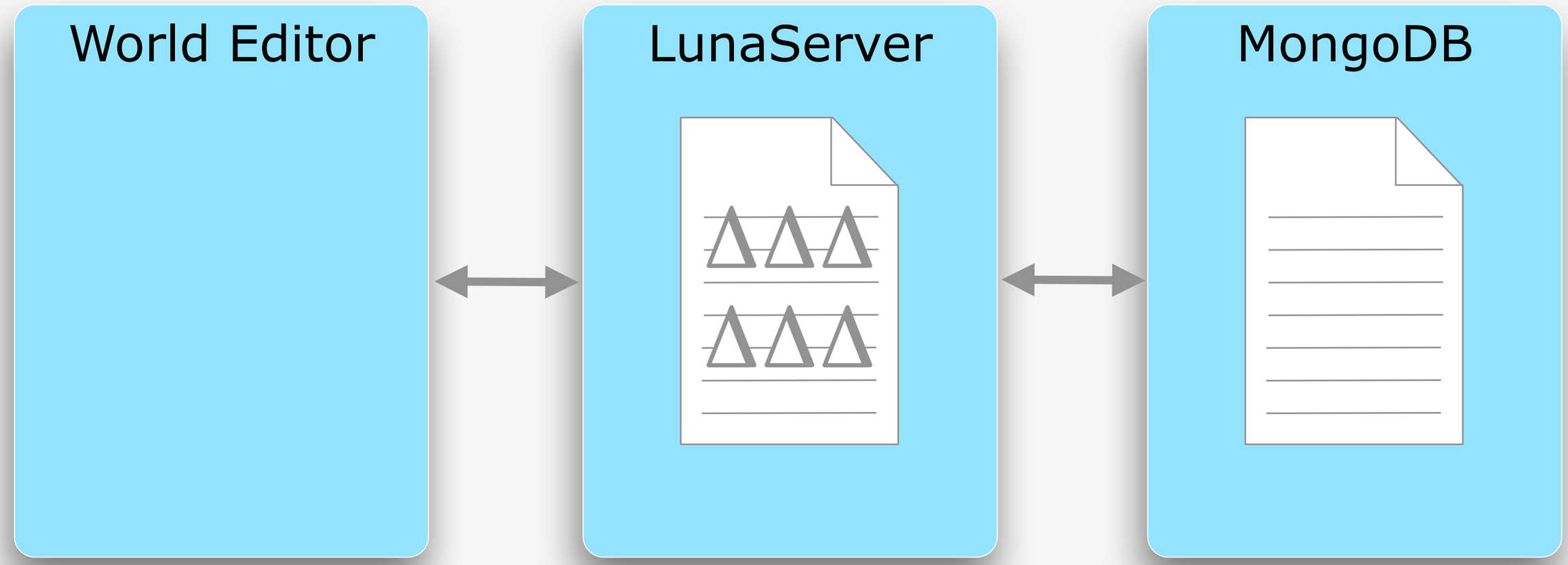
Save never



In the client-server architecture that we developed for our tools, this works quite differently. The editor application doesn't store the document. It is kept on a server. We call it LunaServer.
LunaServer is a process running on the same host PC as the editor. The editor and the server are on the same machine. We have not found it necessary yet to connect over the network.
The application only keeps a cache copy in memory. It does this only so that it can display the document, and process mouse clicks and so on. The authoritative document is maintained by the server.
The server is completely generic. It knows nothing of the nature of the data that it maintains. It just stores the data, collects changes from the client, reports changes to the client when they come in from other clients. More about that in a moment.

[CLICK] While you are using the editor application, it sends every individual change to the server, immediately. The document in its entirety is only ever transmitted once, from the server to the application, when the application starts up. So it can initialize its local cache version.

Save never

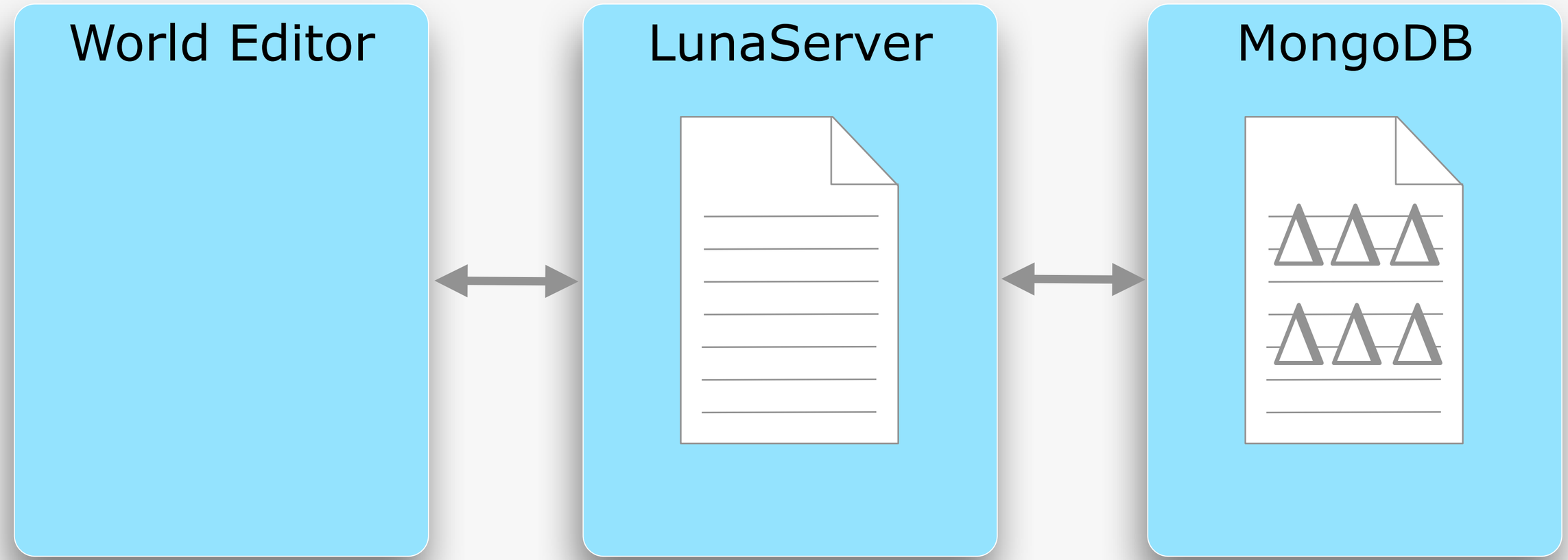


And the server itself is backed by MongoDB. We use MongoDB in several other places in our production pipeline, and it has been serving us very well.

[CLICK] And oops. There it goes again. The editor application has crashed. Not to worry. Your data is safe. Not only all your changes right up to the point where the crash occurred, but also the undo queue and copy/paste data. You can restart the application, and continue working as if nothing ever happened.

Of course the server itself could crash, and so could MongoDB. But that is much less likely. The idea is that the server stabilizes and matures early on, while the editor application continues to be in development.

Save never

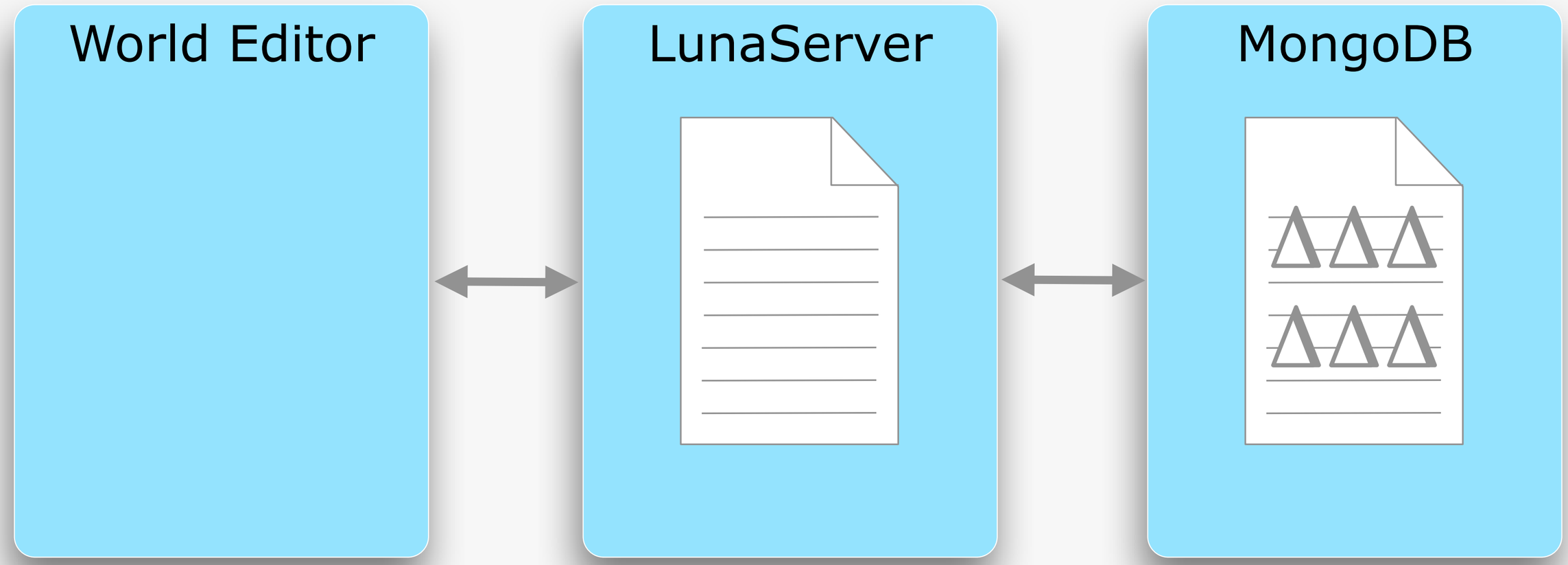


And the server itself is backed by MongoDB. We use MongoDB in several other places in our production pipeline, and it has been serving us very well.

[CLICK] And oops. There it goes again. The editor application has crashed. Not to worry. Your data is safe. Not only all your changes right up to the point where the crash occurred, but also the undo queue and copy/paste data. You can restart the application, and continue working as if nothing ever happened.

Of course the server itself could crash, and so could MongoDB. But that is much less likely. The idea is that the server stabilizes and matures early on, while the editor application continues to be in development.

Save never

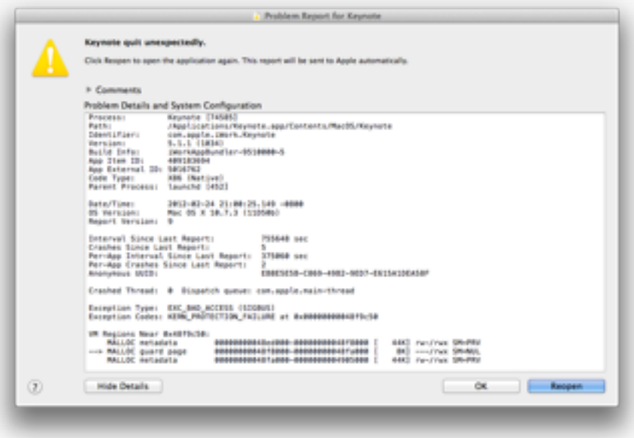


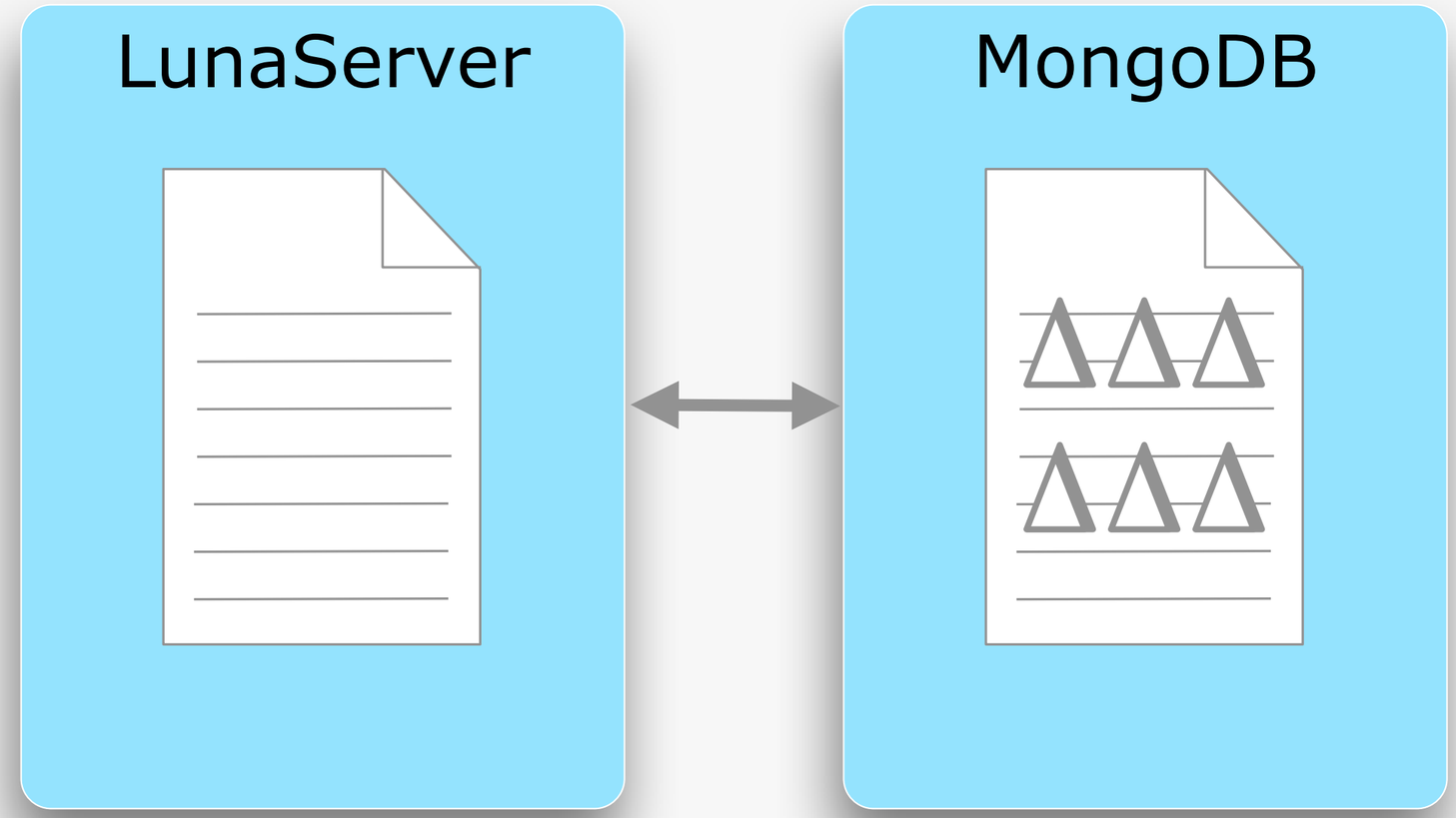
And the server itself is backed by MongoDB. We use MongoDB in several other places in our production pipeline, and it has been serving us very well.

[CLICK] And oops. There it goes again. The editor application has crashed. Not to worry. Your data is safe. Not only all your changes right up to the point where the crash occurred, but also the undo queue and copy/paste data. You can restart the application, and continue working as if nothing ever happened.

Of course the server itself could crash, and so could MongoDB. But that is much less likely. The idea is that the server stabilizes and matures early on, while the editor application continues to be in development.

Save never



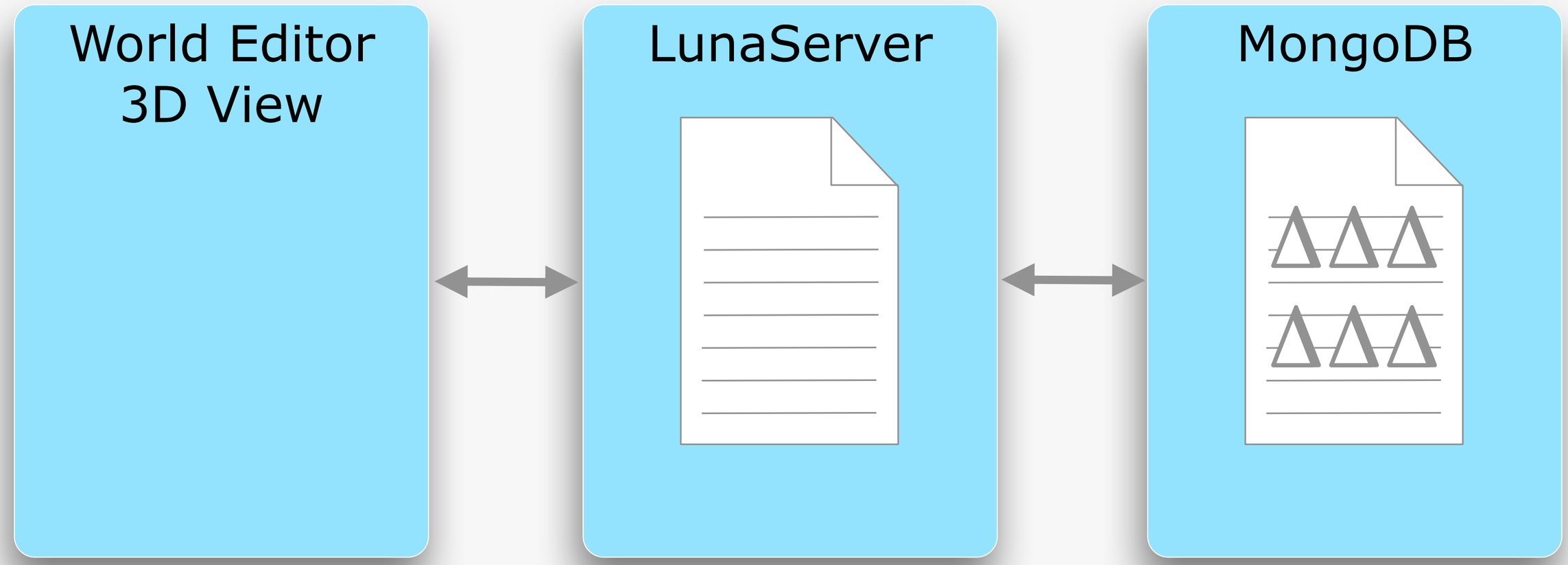


And the server itself is backed by MongoDB. We use MongoDB in several other places in our production pipeline, and it has been serving us very well.

[CLICK] And oops. There it goes again. The editor application has crashed. Not to worry. Your data is safe. Not only all your changes right up to the point where the crash occurred, but also the undo queue and copy/paste data. You can restart the application, and continue working as if nothing ever happened.

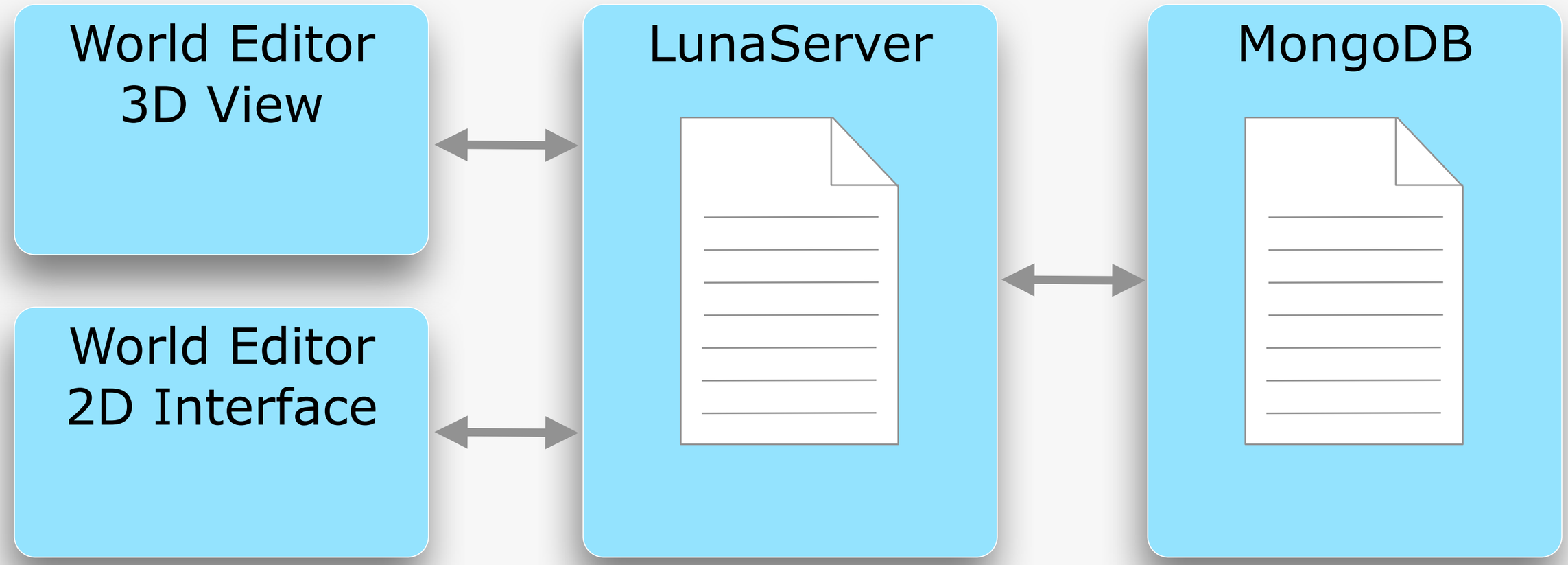
Of course the server itself could crash, and so could MongoDB. But that is much less likely. The idea is that the server stabilizes and matures early on, while the editor application continues to be in development.

Save never



I mentioned a moment ago that the server transmits changes coming from other clients. Our world editor is in fact two clients.
[CLICK] The 3D view editor is separate from the 2D interface.
[CLICK] The 3D view is in fact a completely separate application. It has no 2D interface at all. No HUD, no controls, other than the 3D manipulators. It is written in C++, it is built with our game engine, and it renders to a client window in Chrome. The 2D UI is written entirely in JavaScript and HTML5.
The 3D view and 2D UI never talk to each other. Each only ever communicates with the server. They report their changes to the server, and the server transmits changes to other clients.

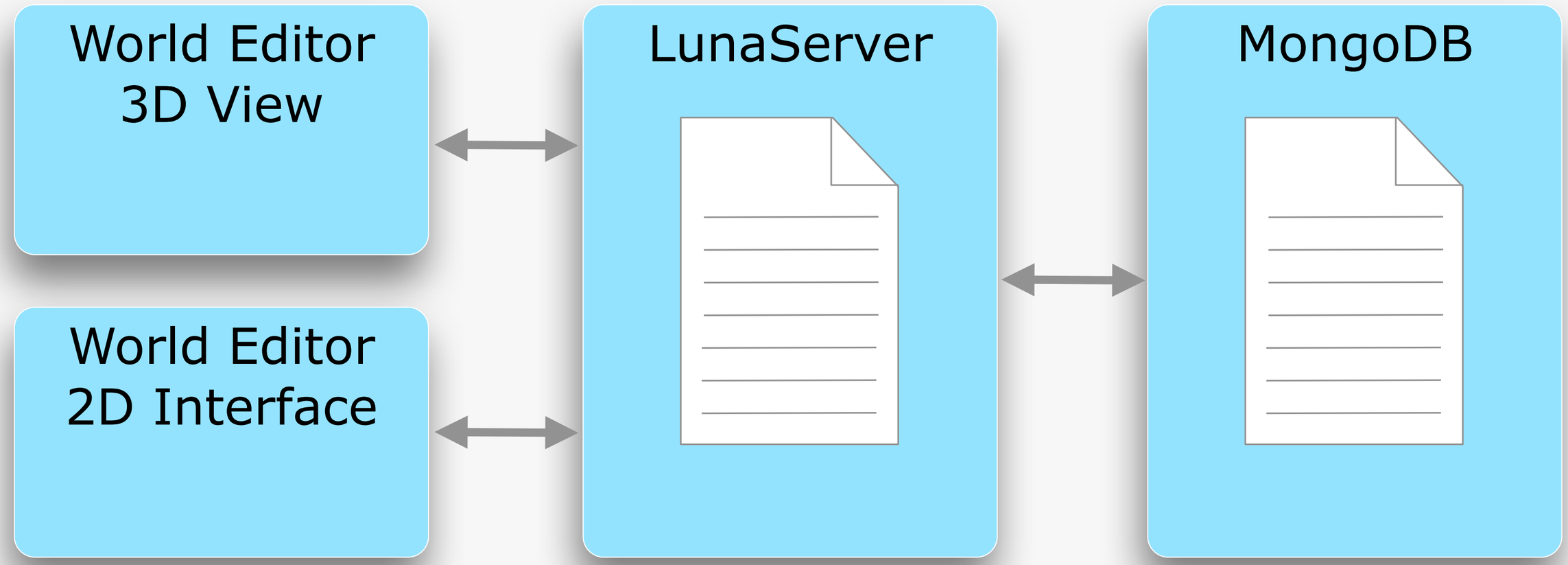
Save never



I mentioned a moment ago that the server transmits changes coming from other clients. Our world editor is in fact two clients.
[CLICK] The 3D view editor is separate from the 2D interface.
[CLICK] The 3D view is in fact a completely separate application. It has no 2D interface at all. No HUD, no controls, other than the 3D manipulators. It is written in C++, it is built with our game engine, and it renders to a client window in Chrome. The 2D UI is written entirely in JavaScript and HTML5.
The 3D view and 2D UI never talk to each other. Each only ever communicates with the server. They report their changes to the server, and the server transmits changes to other clients.

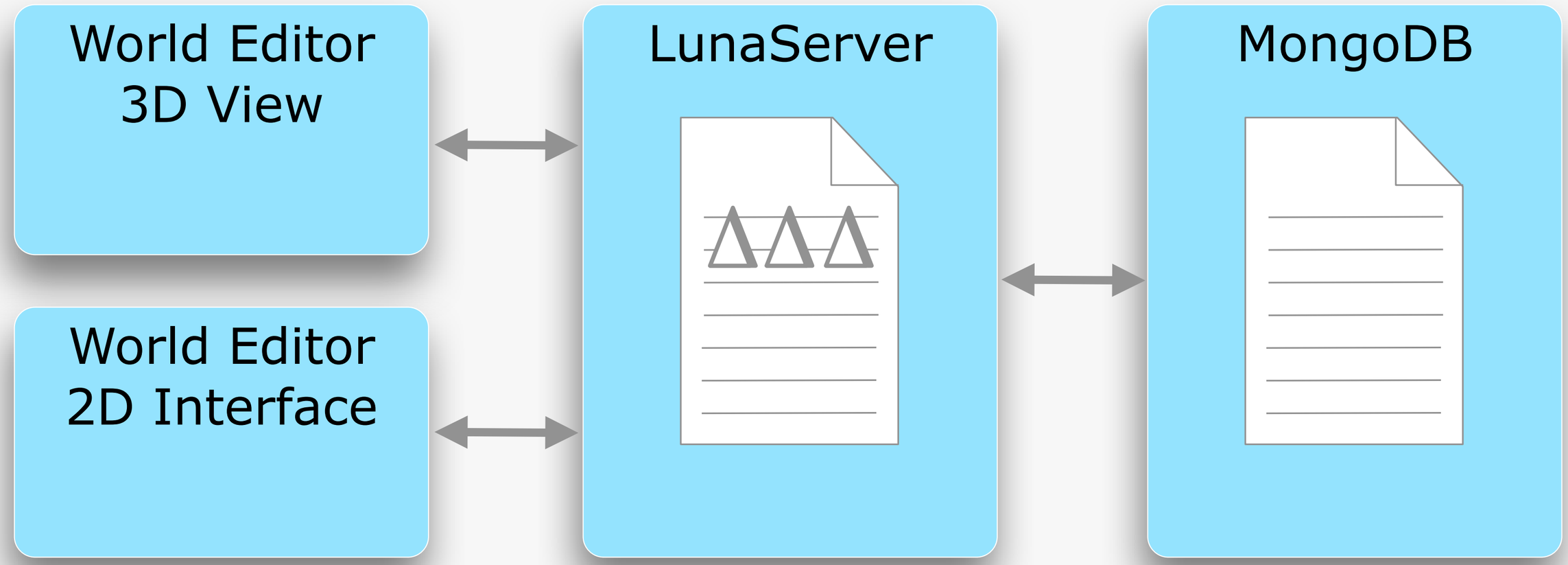
I mentioned a moment ago that the server transmits changes coming from other clients. Our world editor is in fact two clients.
[CLICK] The 3D view editor is separate from the 2D interface.
[CLICK] The 3D view is in fact a completely separate application. It has no 2D interface at all. No HUD, no controls, other than the 3D manipulators. It is written in C++, it is built with our game engine, and it renders to a client window in Chrome. The 2D UI is written entirely in JavaScript and HTML5.
The 3D view and 2D UI never talk to each other. Each only ever communicates with the server. They report their changes to the server, and the server transmits changes to other clients.

Save never



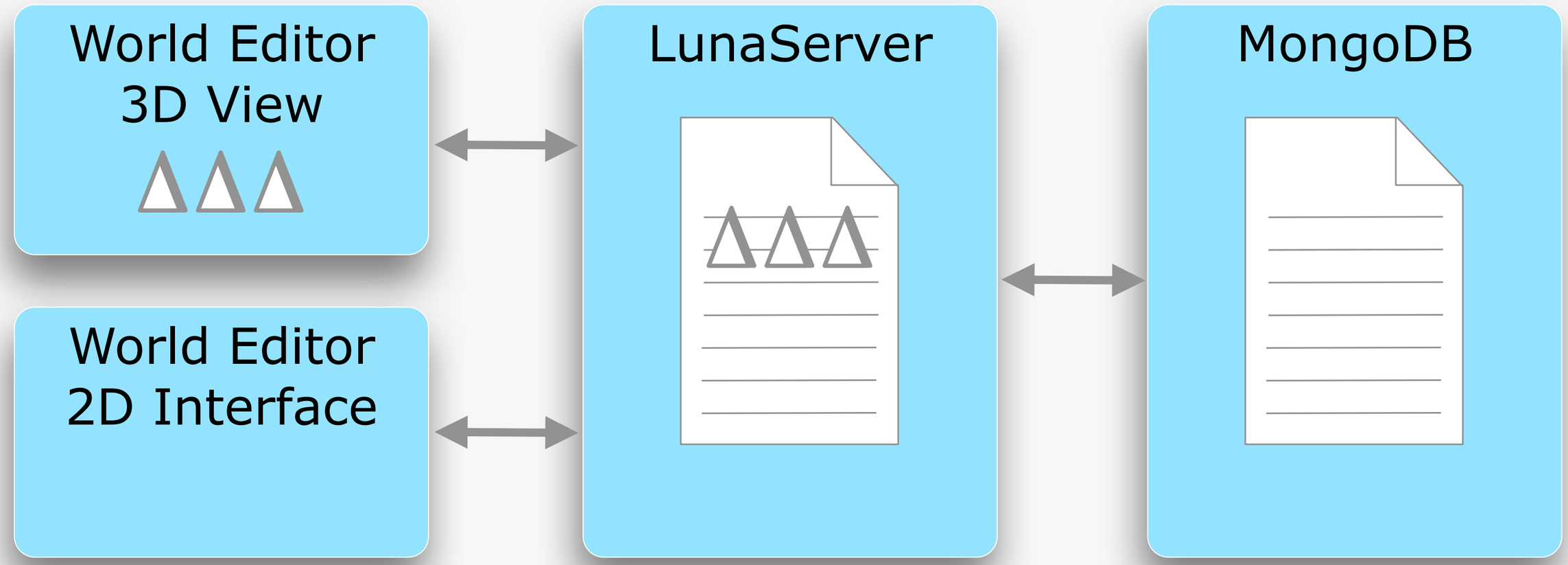
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



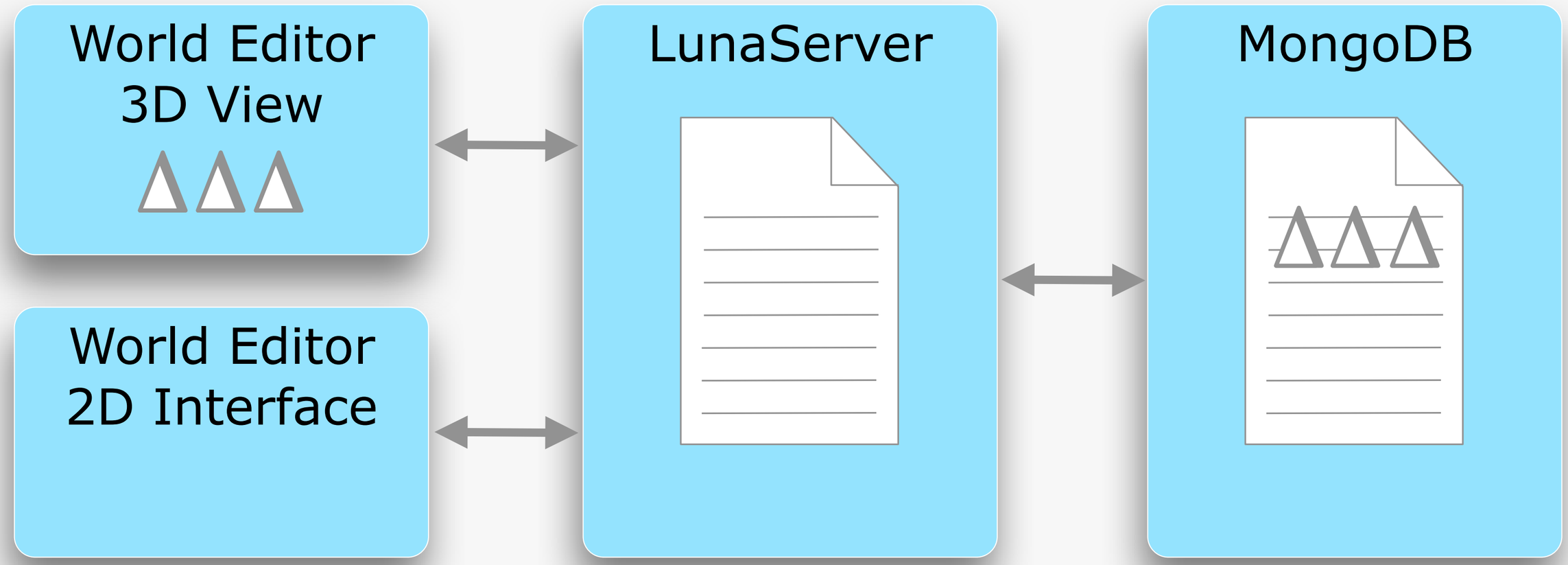
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



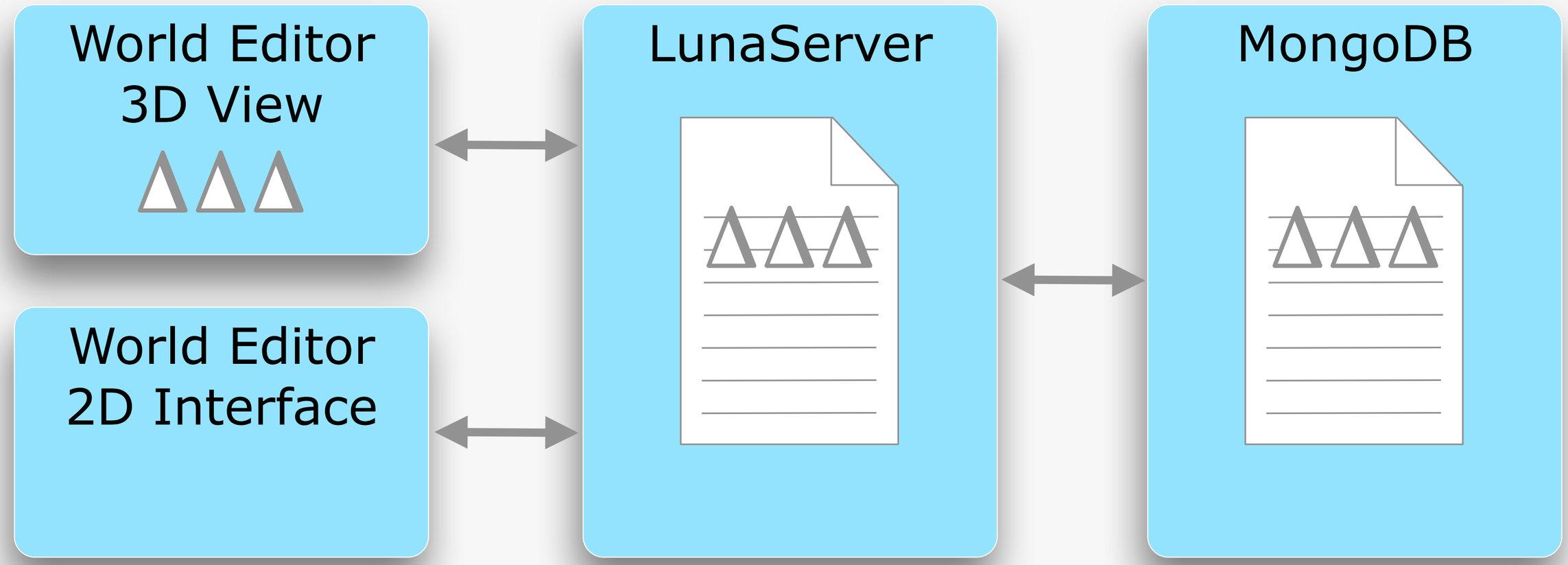
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



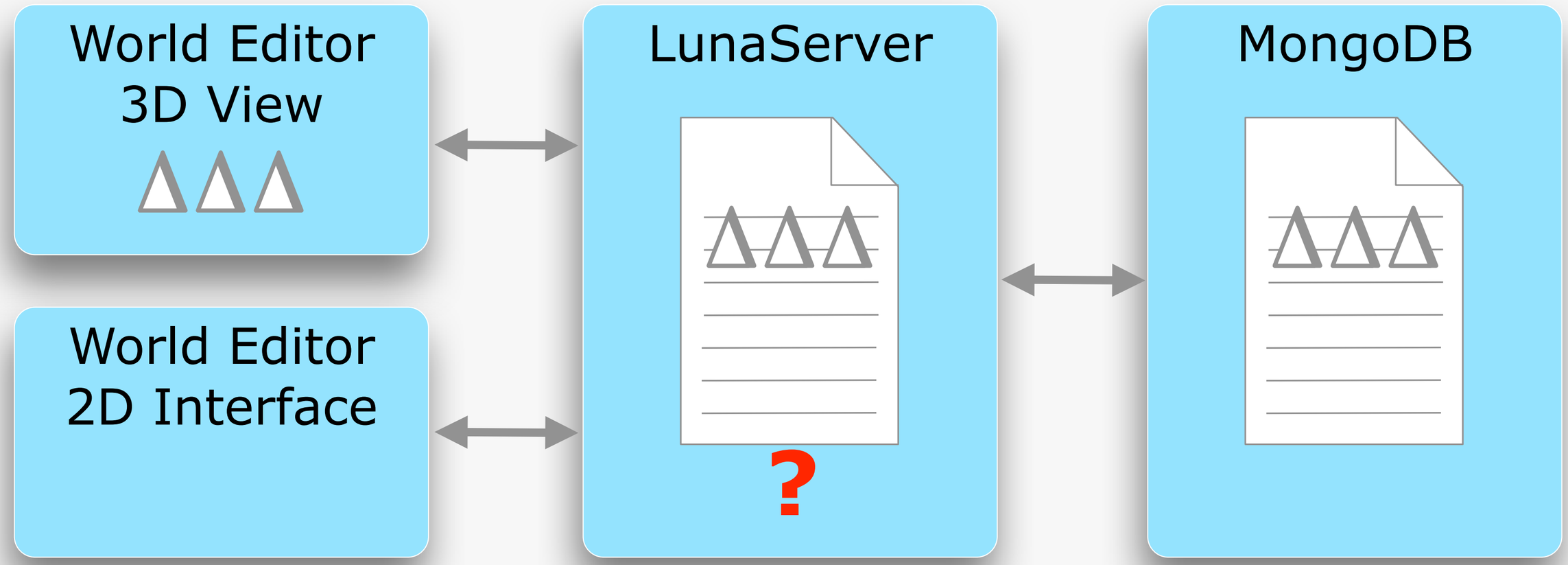
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



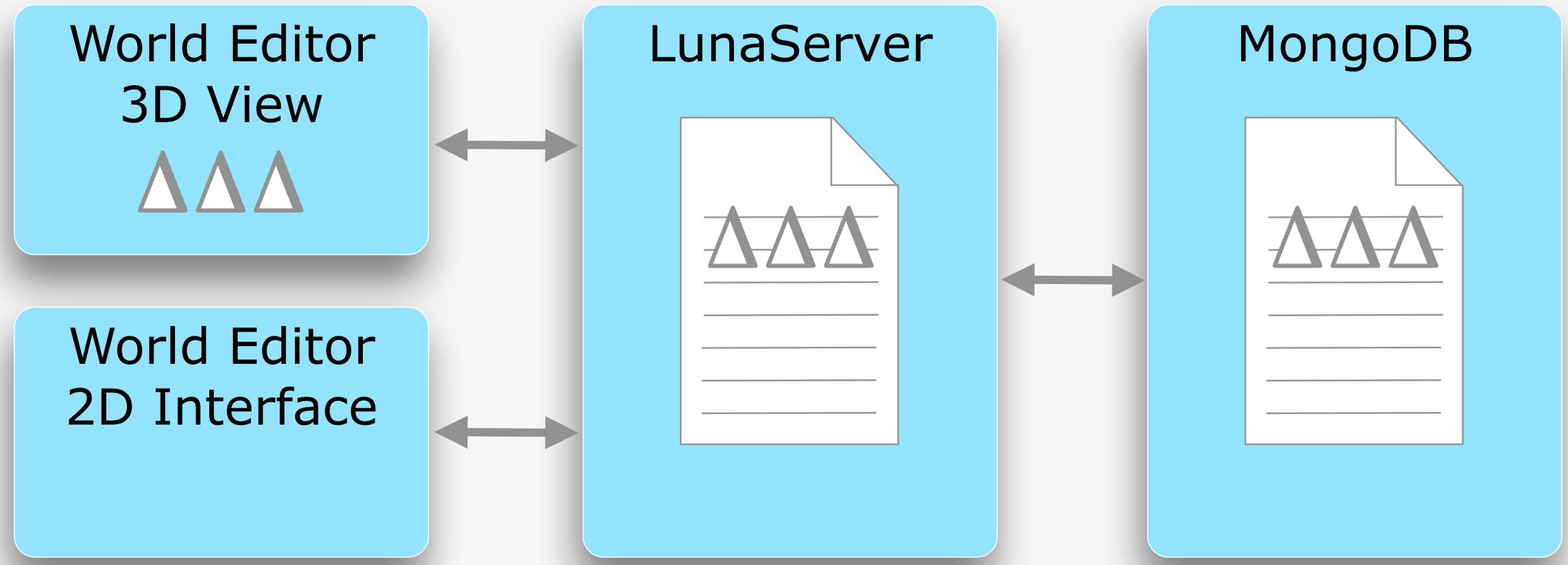
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



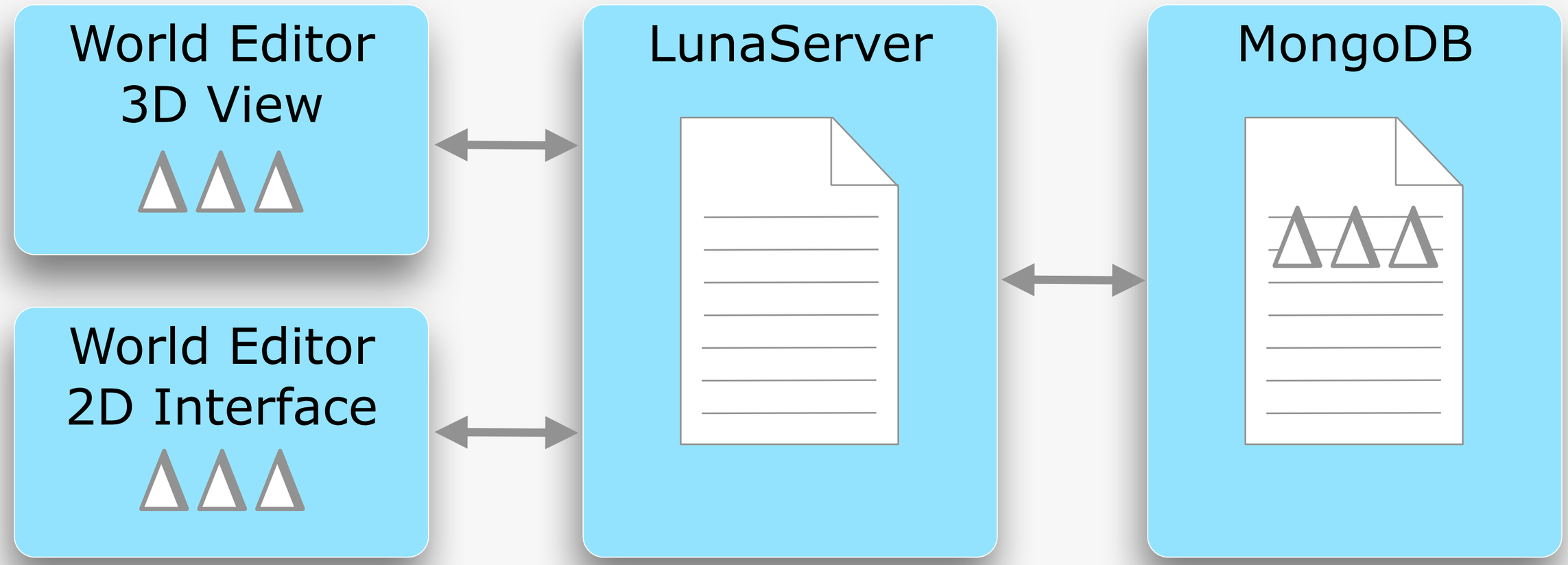
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



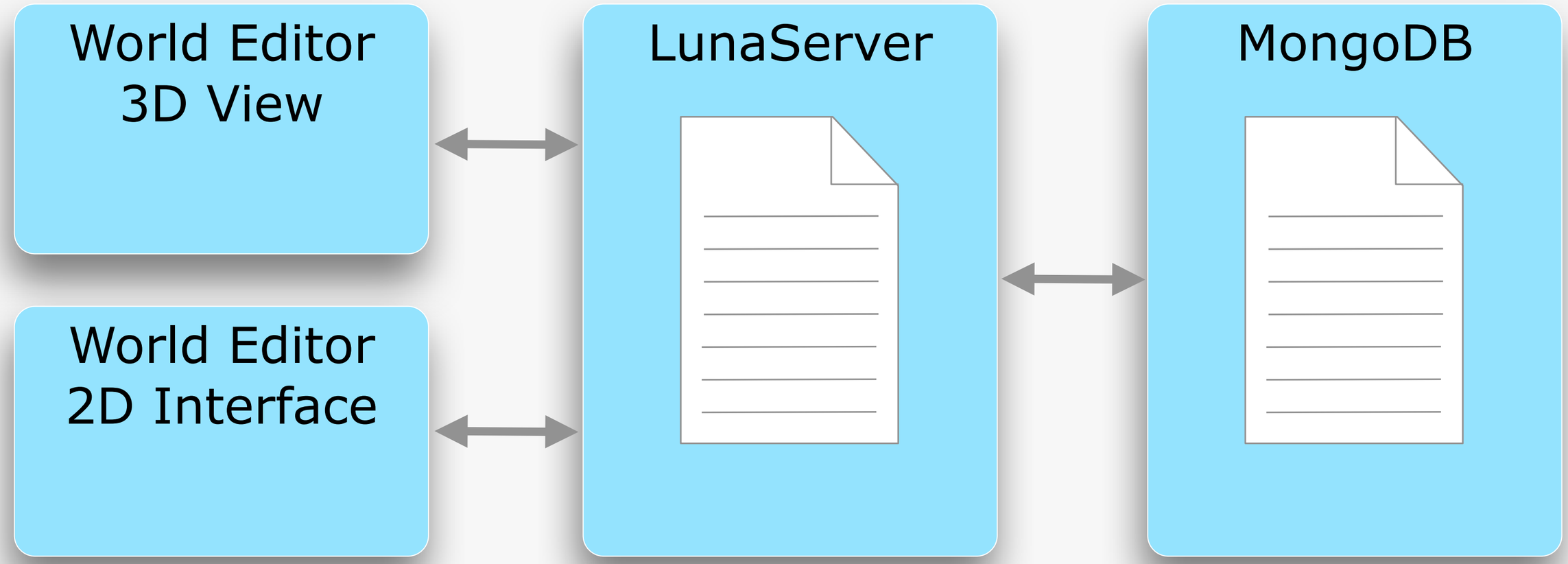
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



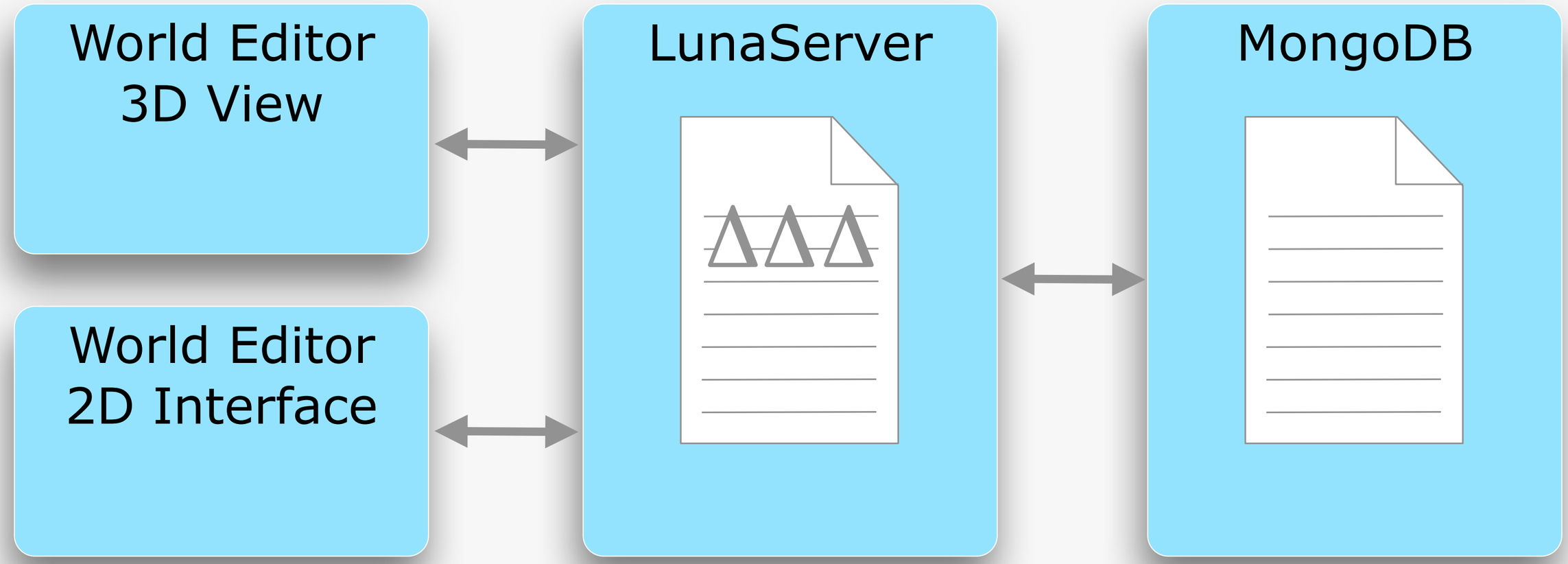
[CLICK] So any changes made in the 3D view are immediately transmitted to the server
[CLICK] The server updates MongoDB
[CLICK] We use a restful client/server model, so in order to keep synchronization with the server, the 2D interface client has to poll the server for changes.

Save never



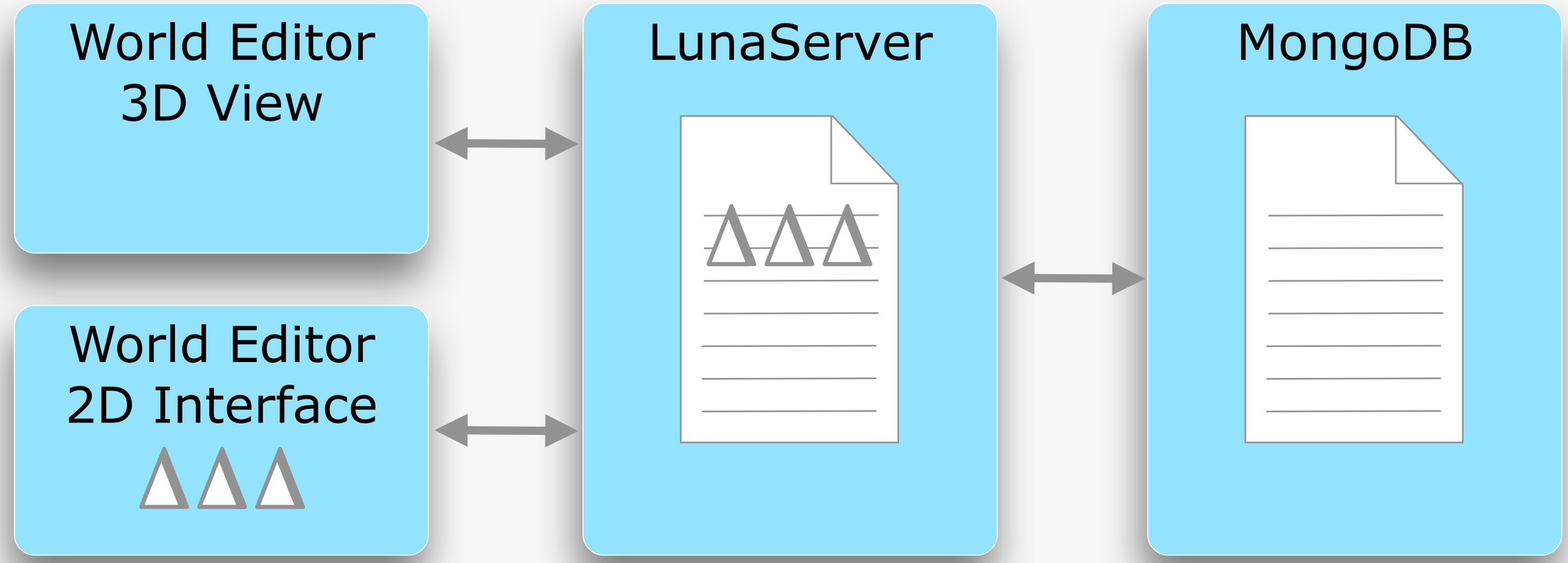
And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

Save never



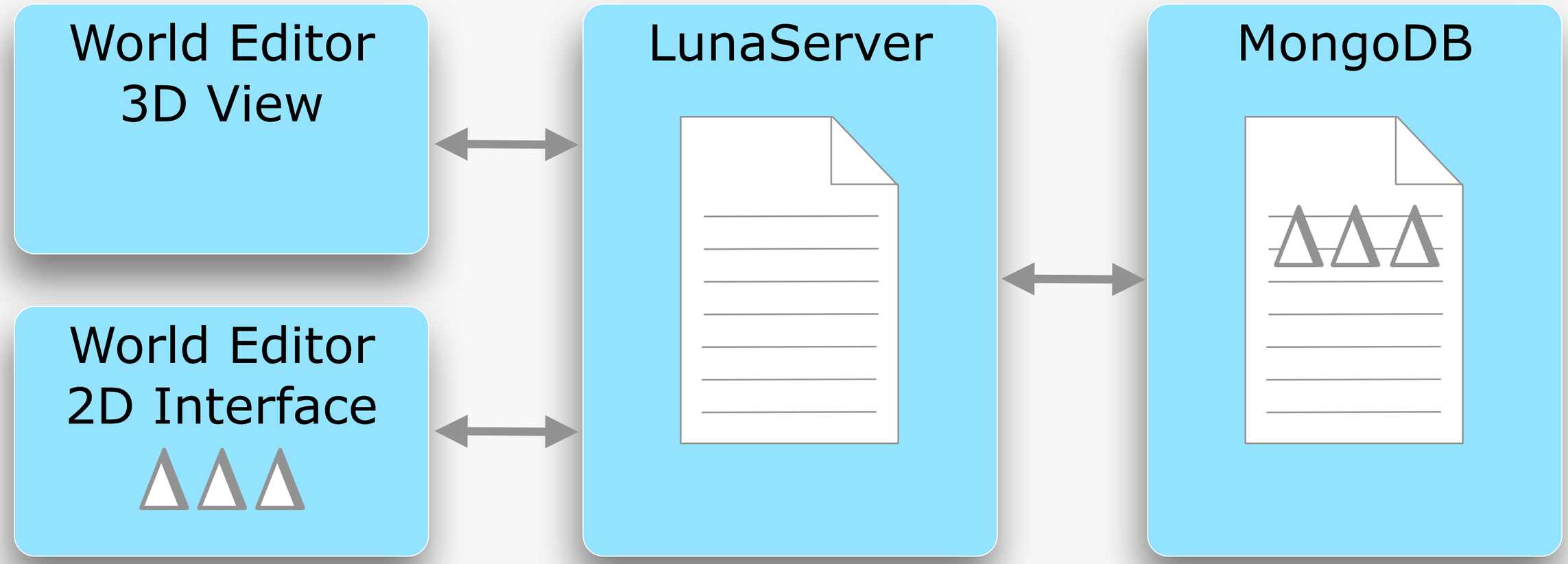
And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

Save never



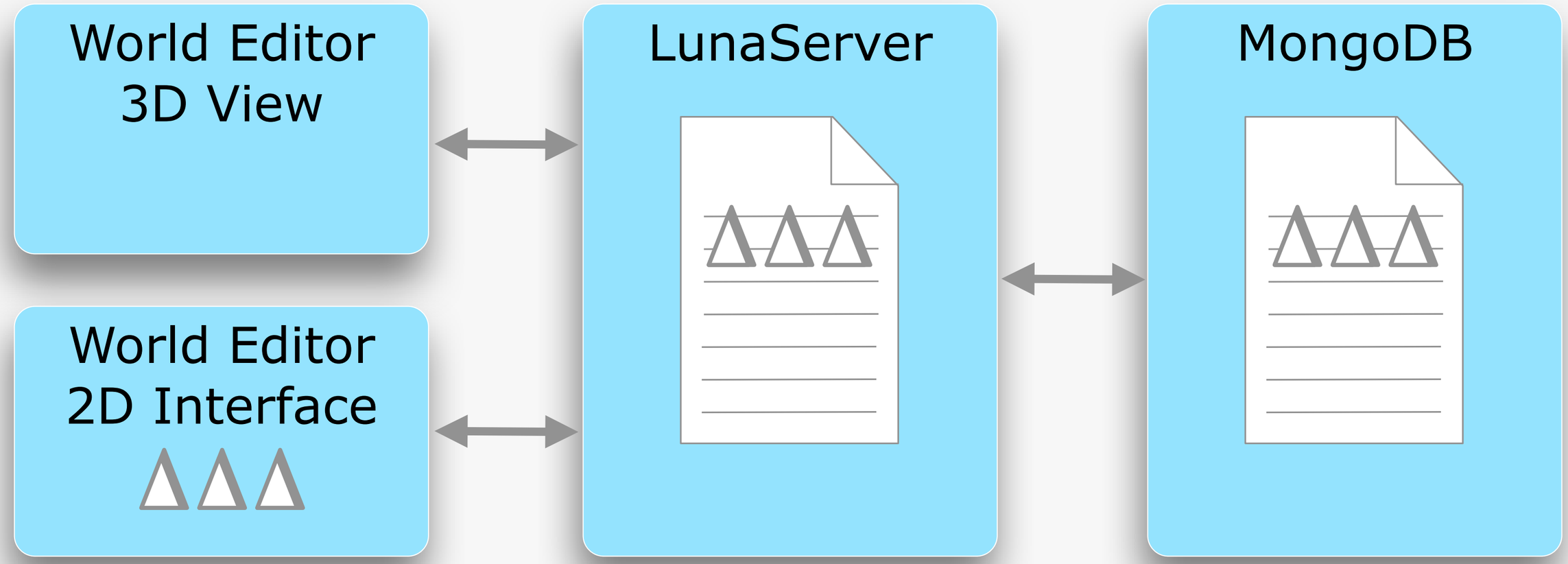
And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

Save never



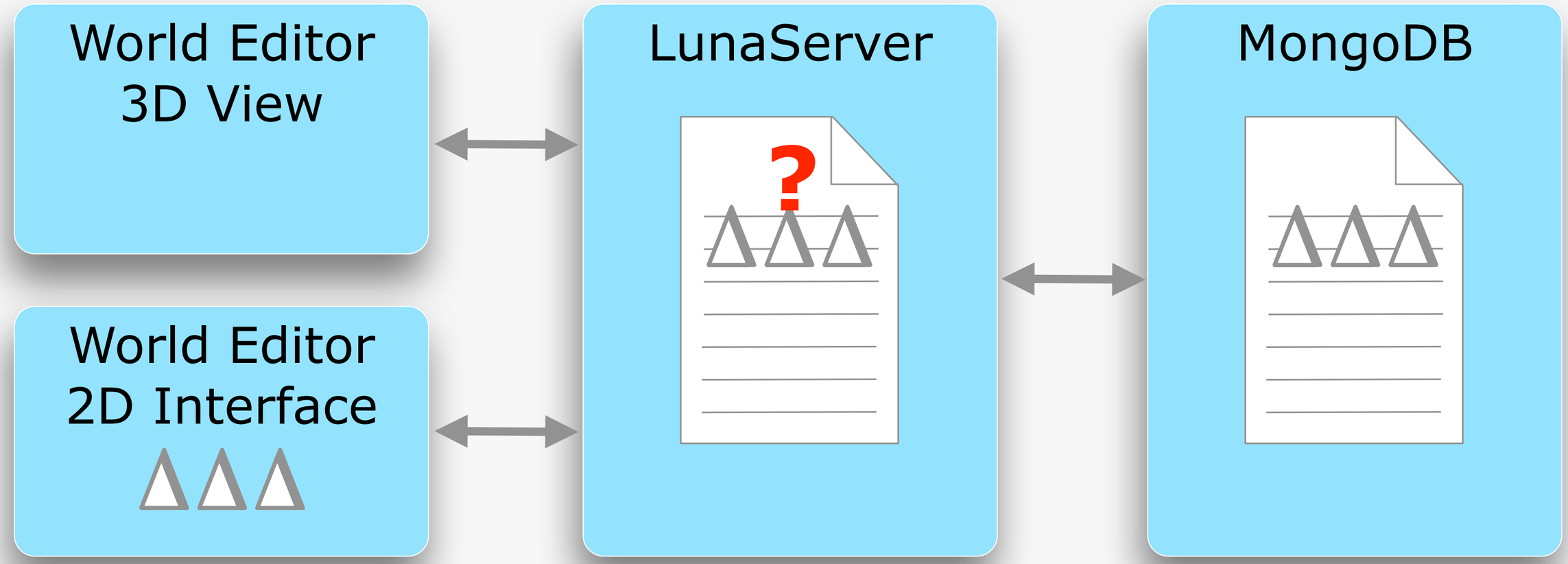
And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

Save never



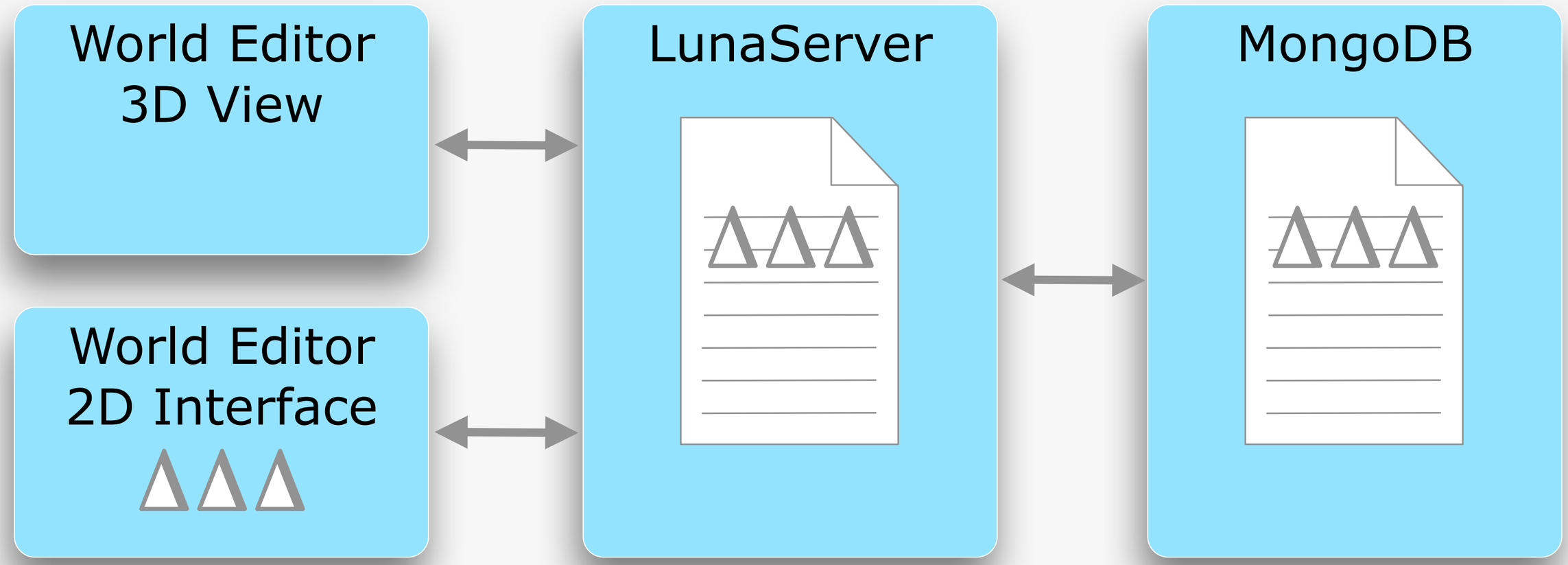
And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

Save never



And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

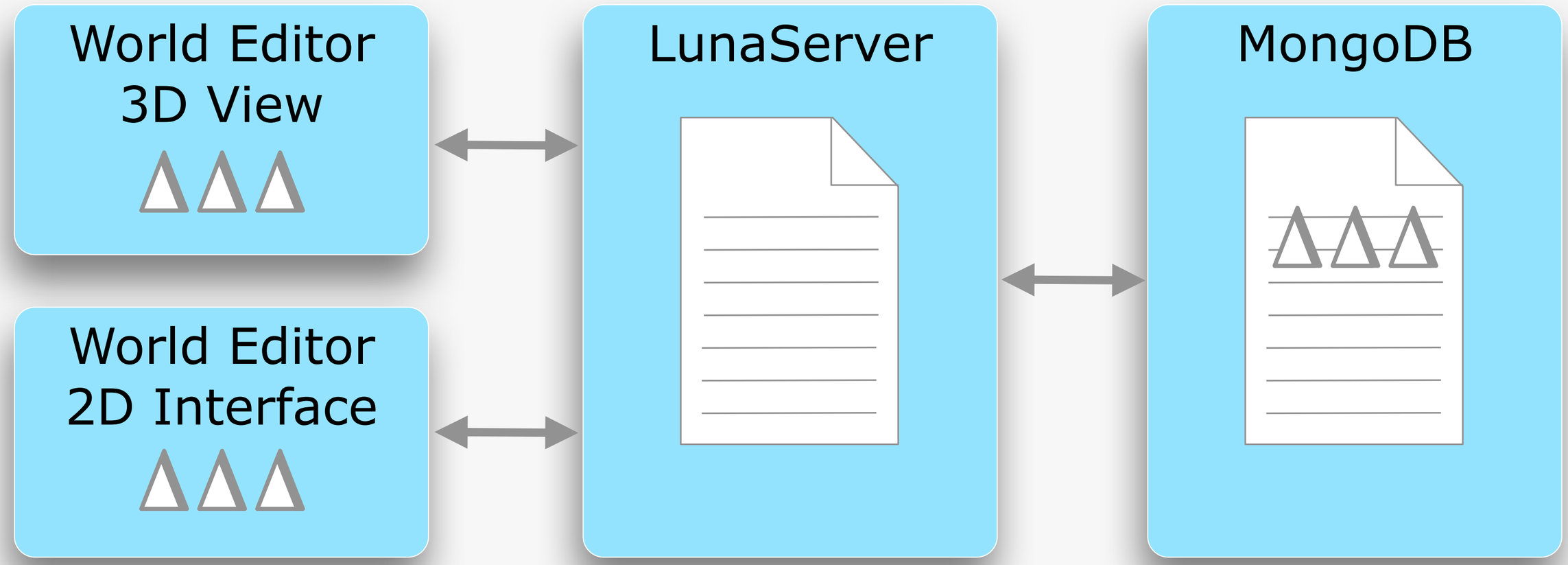
Save never



And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

As I mentioned, it is the gift that keeps on giving. Why stop at two clients?
[CLICK] We could add another 3D view, perhaps showing a completely different part of the level.
[CLICK] And why stop at the world editor? As I mentioned, LunaServer is oblivious to the meaning of the data that it manages. It only deals with documents, changes, poll requests from clients, and MongoDB. All our other tools store their document in the same way.
And the cool thing is, if any document is shared between two editors, changes in one will show up immediately in the other. So if you change an effect in the particle editor, and that effect has been placed in a level, and you are viewing it in the world editor, any changes you make in the particle editor have immediate effect in the world editor's 3D view.

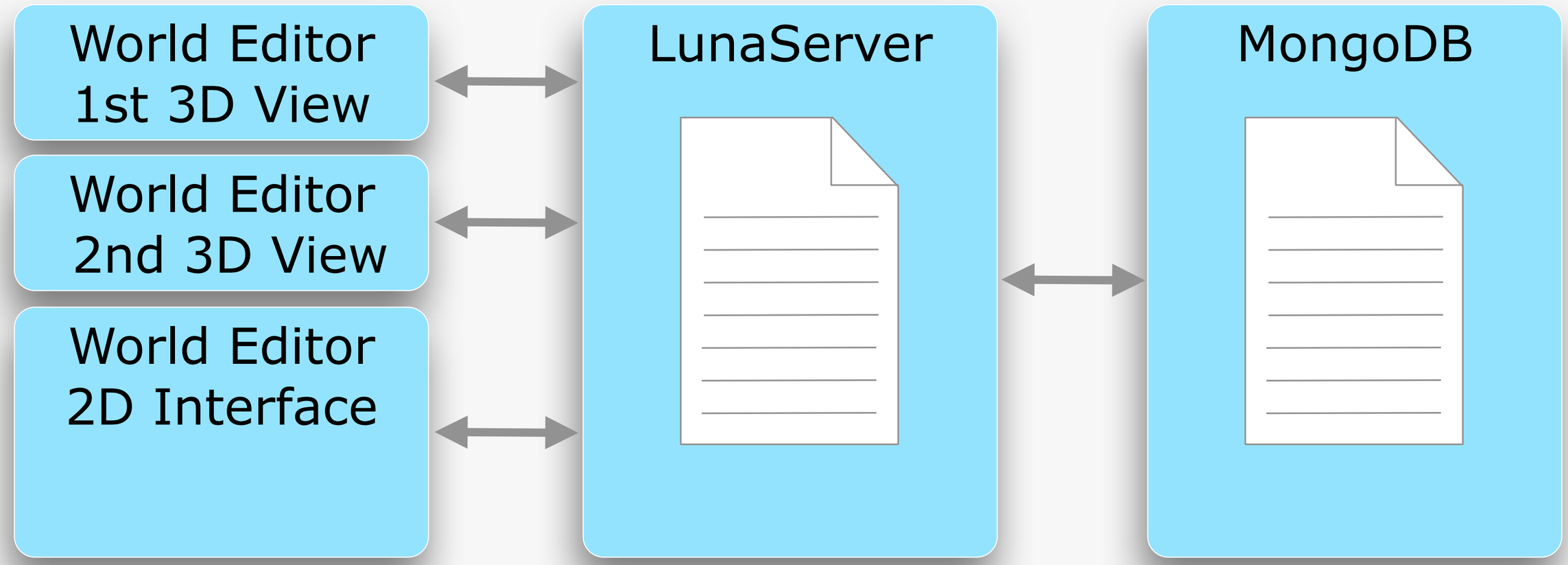
Save never



And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

As I mentioned, it is the gift that keeps on giving. Why stop at two clients?
[CLICK] We could add another 3D view, perhaps showing a completely different part of the level.
[CLICK] And why stop at the world editor? As I mentioned, LunaServer is oblivious to the meaning of the data that it manages. It only deals with documents, changes, poll requests from clients, and MongoDB. All our other tools store their document in the same way.
And the cool thing is, if any document is shared between two editors, changes in one will show up immediately in the other. So if you change an effect in the particle editor, and that effect has been placed in a level, and you are viewing it in the world editor, any changes you make in the particle editor have immediate effect in the world editor's 3D view.

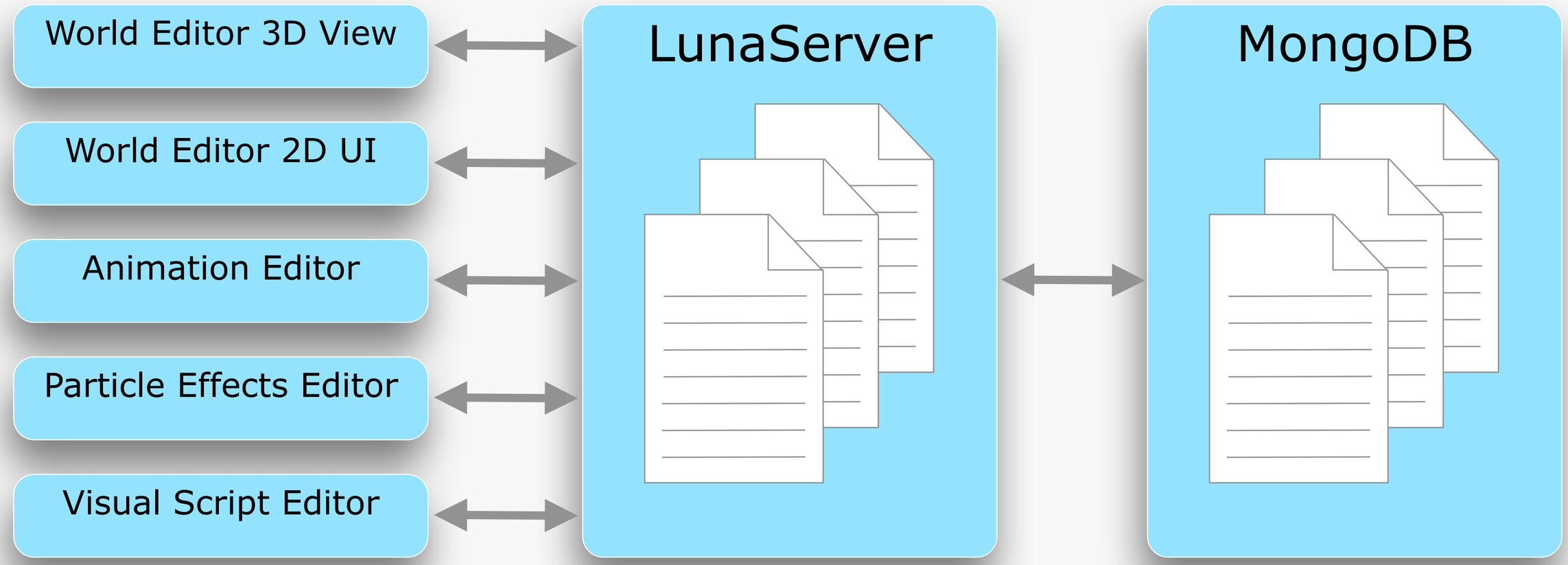
Save never



And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

As I mentioned, it is the gift that keeps on giving. Why stop at two clients?
[CLICK] We could add another 3D view, perhaps showing a completely different part of the level.
[CLICK] And why stop at the world editor? As I mentioned, LunaServer is oblivious to the meaning of the data that it manages. It only deals with documents, changes, poll requests from clients, and MongoDB. All our other tools store their document in the same way.
And the cool thing is, if any document is shared between two editors, changes in one will show up immediately in the other. So if you change an effect in the particle editor, and that effect has been placed in a level, and you are viewing it in the world editor, any changes you make in the particle editor have immediate effect in the world editor's 3D view.

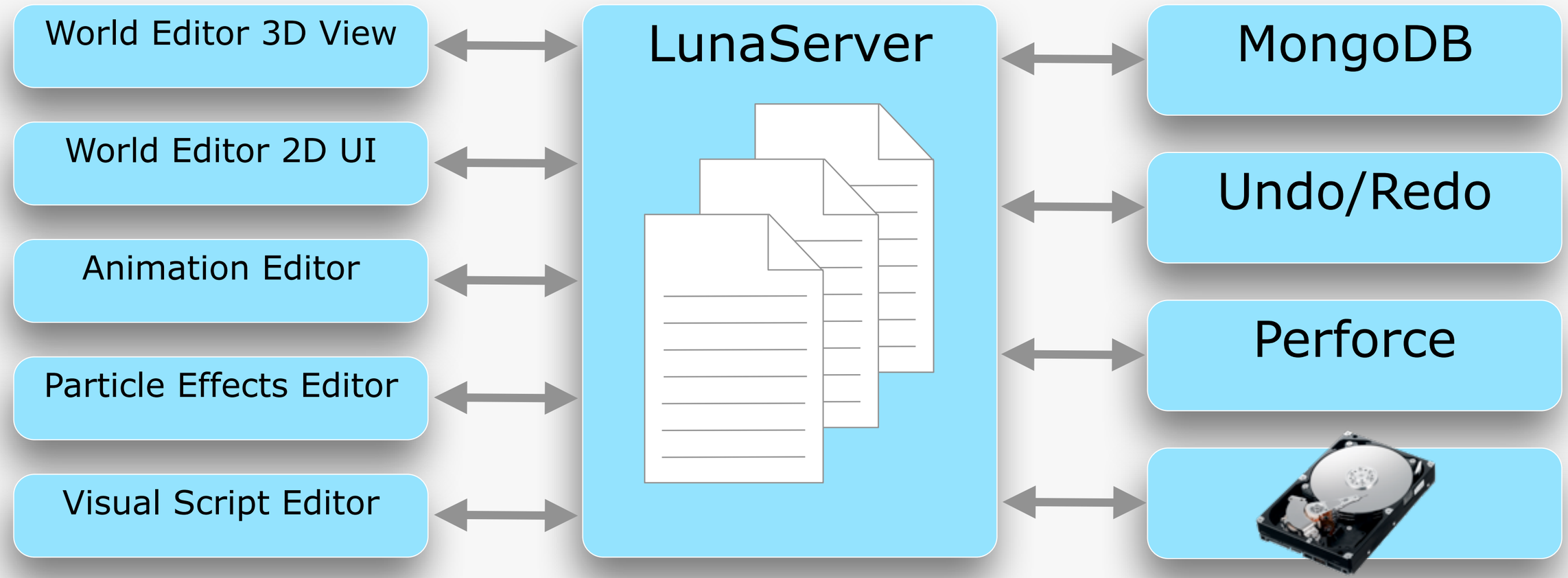
Save never



And vice versa, changes made in the 2D UI are transmitted immediately to the server, and the next time the 3D view polls for changes, it will get the updates.

As I mentioned, it is the gift that keeps on giving. Why stop at two clients?
[CLICK] We could add another 3D view, perhaps showing a completely different part of the level.
[CLICK] And why stop at the world editor? As I mentioned, LunaServer is oblivious to the meaning of the data that it manages. It only deals with documents, changes, poll requests from clients, and MongoDB. All our other tools store their document in the same way.
And the cool thing is, if any document is shared between two editors, changes in one will show up immediately in the other. So if you change an effect in the particle editor, and that effect has been placed in a level, and you are viewing it in the world editor, any changes you make in the particle editor have immediate effect in the world editor's 3D view.

Save never



And it keeps on giving. Because LunaServer already manages documents in a database, and deals with document changes and synchronization on a very abstract level, this is the perfect place to implement an undo system. It manages undo by keeping a log of all database transactions, it computes the inverse, so that when told to perform an undo, it generates a change to revert the document in question to an earlier state. The clients, the editors, don't need to do anything special. They just process their the changes they receive in the normal way. There is no undo code at all in any of the editors.

And Perforce integration has also been implemented on server level. There is no code at all in any of the editors, to deal with Perforce. Same with loading and saving to disk, and reverting. All prompts and dialogs are handled on the server level.

This simplifies editor code significantly, and offers the user a very consistent look and behavior.

Client/server tools architecture

So there you have it.
LunaServer matures early on, because it is a general purpose document manager that is already feature complete.
[CLICK] This model offers protection against data loss due to crashes
[CLICK] Synchronization between multiple editors is a snap, just by following the rules of synchronizing with the server
[CLICK] Editors are written only to do their specialized job
[CLICK] And all undo, Perforce integration and so on is automatically available in any tool, and this offers the user a consistent experience across tools, and saves editor programmers a ton of work.

Client/server tools architecture

- Server matures before editors do

So there you have it.
LunaServer matures early on, because it is a general purpose document manager that is already feature complete.
[CLICK] This model offers protection against data loss due to crashes
[CLICK] Synchronization between multiple editors is a snap, just by following the rules of synchronizing with the server
[CLICK] Editors are written only to do their specialized job
[CLICK] And all undo, Perforce integration and so on is automatically available in any tool, and this offers the user a consistent experience across tools, and saves editor programmers a ton of work.

Client/server tools architecture

- Server matures before editors do
- Prevent loss of work in case of crash

So there you have it.
LunaServer matures early on, because it is a general purpose document manager that is already feature complete.
[CLICK] This model offers protection against data loss due to crashes
[CLICK] Synchronization between multiple editors is a snap, just by following the rules of synchronizing with the server
[CLICK] Editors are written only to do their specialized job
[CLICK] And all undo, Perforce integration and so on is automatically available in any tool, and this offers the user a consistent experience across tools, and saves editor programmers a ton of work.

Client/server tools architecture

- Server matures before editors do
- Prevent loss of work in case of crash
- Changes in one editor immediately visible in another

So there you have it.
LunaServer matures early on, because it is a general purpose document manager that is already feature complete.
[CLICK] This model offers protection against data loss due to crashes
[CLICK] Synchronization between multiple editors is a snap, just by following the rules of synchronizing with the server
[CLICK] Editors are written only to do their specialized job
[CLICK] And all undo, Perforce integration and so on is automatically available in any tool, and this offers the user a consistent experience across tools, and saves editor programmers a ton of work.

Client/server tools architecture

- Server matures before editors do
- Prevent loss of work in case of crash
- Changes in one editor immediately visible in another
- Simplified editor architecture

So there you have it.
LunaServer matures early on, because it is a general purpose document manager that is already feature complete.
[CLICK] This model offers protection against data loss due to crashes
[CLICK] Synchronization between multiple editors is a snap, just by following the rules of synchronizing with the server
[CLICK] Editors are written only to do their specialized job
[CLICK] And all undo, Perforce integration and so on is automatically available in any tool, and this offers the user a consistent experience across tools, and saves editor programmers a ton of work.

Client/server tools architecture

- Server matures before editors do
- Prevent loss of work in case of crash
- Changes in one editor immediately visible in another
- Simplified editor architecture
- Undo/redo, Perforce integration, save/load/revert is consistent across editors, no additional effort for engineer

So there you have it.
LunaServer matures early on, because it is a general purpose document manager that is already feature complete.
[CLICK] This model offers protection against data loss due to crashes
[CLICK] Synchronization between multiple editors is a snap, just by following the rules of synchronizing with the server
[CLICK] Editors are written only to do their specialized job
[CLICK] And all undo, Perforce integration and so on is automatically available in any tool, and this offers the user a consistent experience across tools, and saves editor programmers a ton of work.

Closing words

Downtime = Number of breakages × Time to fix breakage × Number of people waiting

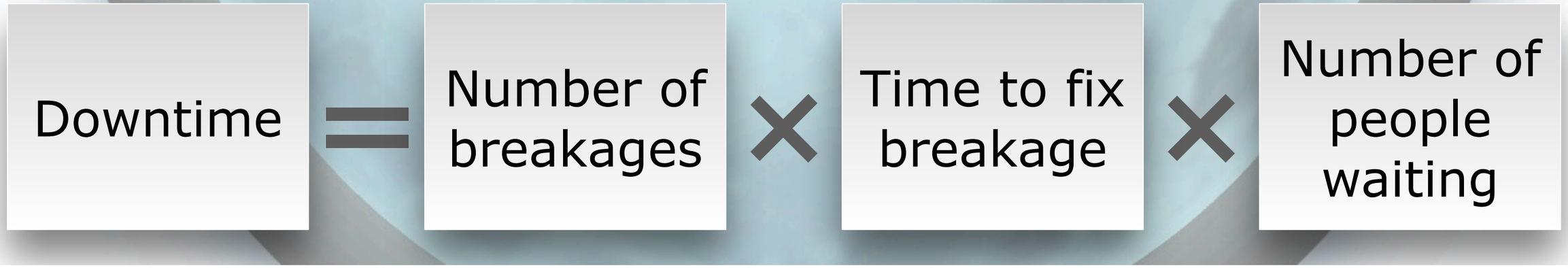
As engineers, we like to design how all our systems to work beautifully and efficiently. But if we concentrate on perfection, we end up with a system that either works perfectly or not at all. No plan B.

[CLICK] We need to consider that while our software is in development, imperfection must be expected, and will be common. How well your production pipeline fails is an important factor in its success.
Allow users to switch to older builds. Make assertions less disruptive. Take measures to prevent data loss in the case of a failure.

[CLICK] A plan that attempts only to reduce failure is doomed to fail. It is the quality of failure that needs to be controlled.

Closing words

- Don’t design your systems for the perfect case



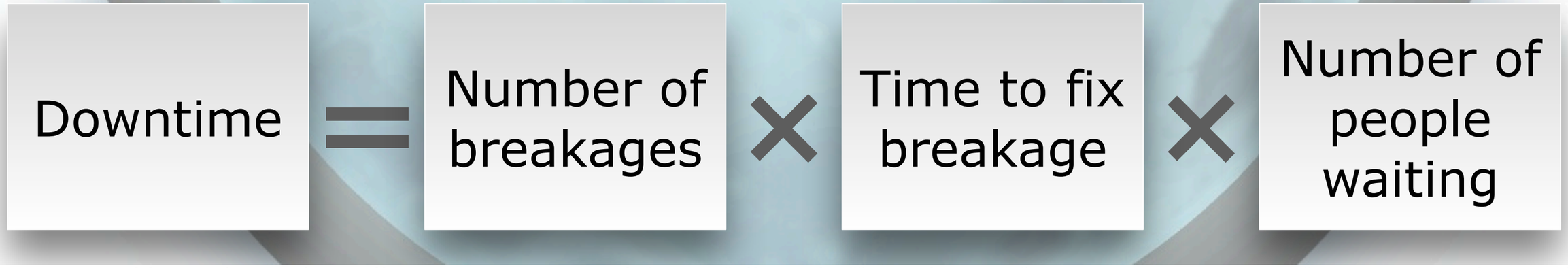
As engineers, we like to design how all our systems to work beautifully and efficiently. But if we concentrate on perfection, we end up with a system that either works perfectly or not at all. No plan B.

[CLICK] We need to consider that while our software is in development, imperfection must be expected, and will be common. How well your production pipeline fails is an important factor in its success.
Allow users to switch to older builds. Make assertions less disruptive. Take measures to prevent data loss in the case of a failure.

[CLICK] A plan that attempts only to reduce failure is doomed to fail. It is the quality of failure that needs to be controlled.

Closing words

- Don’t design your systems for the perfect case
- You need a plan for daily breakage problems



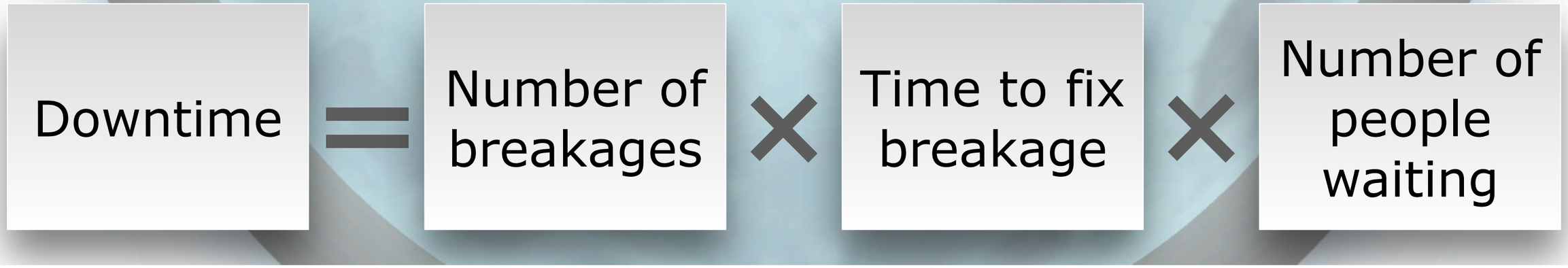
As engineers, we like to design how all our systems to work beautifully and efficiently. But if we concentrate on perfection, we end up with a system that either works perfectly or not at all. No plan B.

[CLICK] We need to consider that while our software is in development, imperfection must be expected, and will be common. How well your production pipeline fails is an important factor in its success.
Allow users to switch to older builds. Make assertions less disruptive. Take measures to prevent data loss in the case of a failure.

[CLICK] A plan that attempts only to reduce failure is doomed to fail. It is the quality of failure that needs to be controlled.

Closing words

- Don’t design your systems for the perfect case
- You need a plan for daily breakage problems
- Prevention alone is not enough



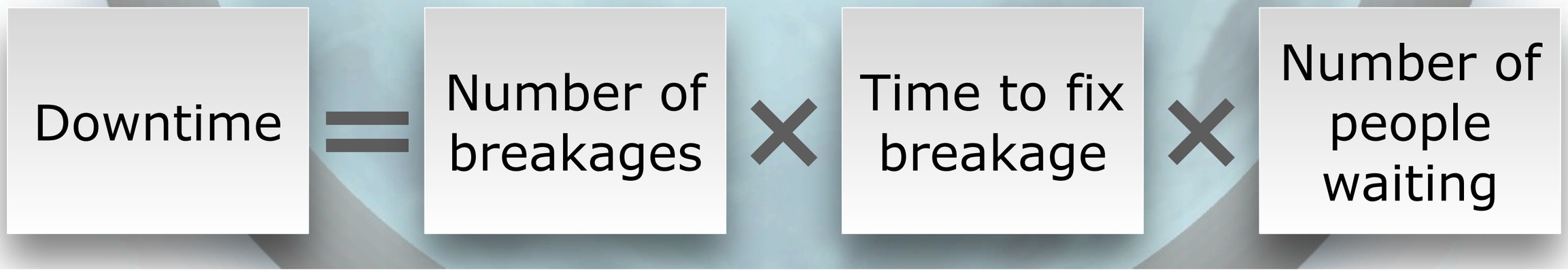
As engineers, we like to design how all our systems to work beautifully and efficiently. But if we concentrate on perfection, we end up with a system that either works perfectly or not at all. No plan B.

[CLICK] We need to consider that while our software is in development, imperfection must be expected, and will be common. How well your production pipeline fails is an important factor in its success.
Allow users to switch to older builds. Make assertions less disruptive. Take measures to prevent data loss in the case of a failure.

[CLICK] A plan that attempts only to reduce failure is doomed to fail. It is the quality of failure that needs to be controlled.

Closing words

- Don’t design your systems for the perfect case
- You need a plan for daily breakage problems
- Prevention alone is not enough
- Remember: **the build <u>will</u> break.** It’s what builds do



As engineers, we like to design how all our systems to work beautifully and efficiently. But if we concentrate on perfection, we end up with a system that either works perfectly or not at all. No plan B.

[CLICK] We need to consider that while our software is in development, imperfection must be expected, and will be common. How well your production pipeline fails is an important factor in its success.
Allow users to switch to older builds. Make assertions less disruptive. Take measures to prevent data loss in the case of a failure.

[CLICK] A plan that attempts only to reduce failure is doomed to fail. It is the quality of failure that needs to be controlled.

http://www.insomniacgames.com/category/research-development/

- Please fill out your survey
- Samples & notes on our website
  (soonish)
- We're hiring!

INSOMNIAC GAMES