# The Art of Destruction in Rainbow Six: Siege

**Julien L'Heureux**
Technical Lead / Physics Programmer
Ubisoft Montreal

# The Context

*How procedural destruction made it into Rainbow Six: Siege*

# What is Procedural Destruction?

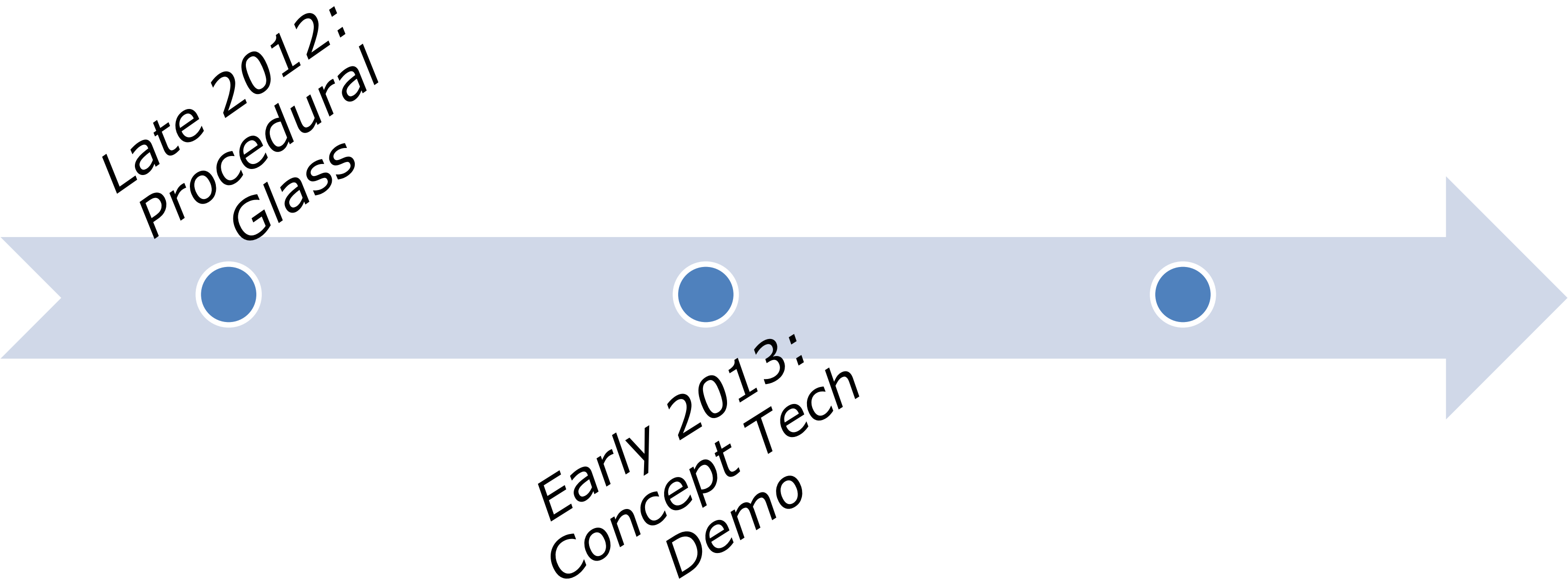
A change in the state of an object **generated at runtime**, where the **outcome is unique**.

In opposition to pre-fragmented destruction, which is pre-determined and has a **fixed outcome**.

# A Brief History

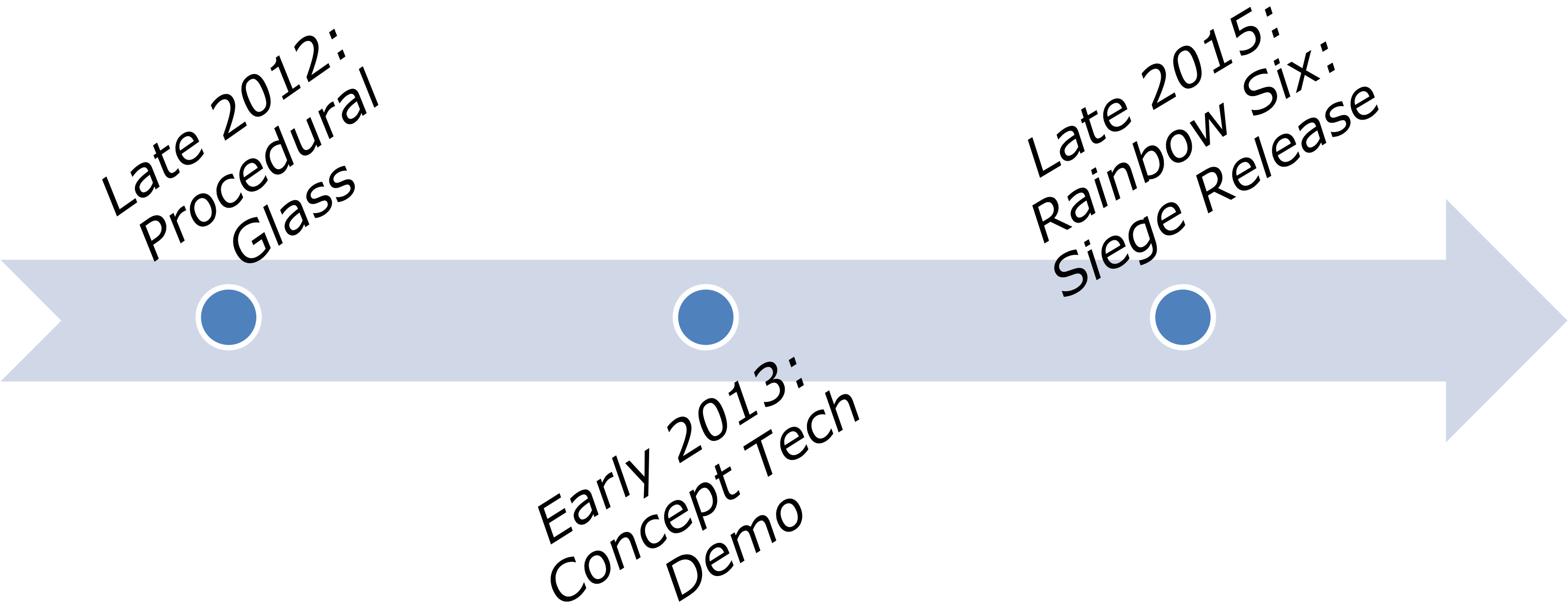
# Procedural Glass Prototype

# A Brief History



# Concept Tech Demo

# A Brief History



# Presentation Outline

# Presentation Outline

I. RealBlast Overview
II. Destruction in Rainbow 6: Siege
III. RealBlast Tech
IV. Performance
V. Online
VI. Conclusion & Future Development

# RealBlast Overview

# REALBLAST

**Realtime Destruction Solution**

- Small team, mostly programmers
- Dedicated to destruction for ~5years
- Part of the TG

*Shout out to Alexandre Ouimet*

# TECHNOLOGY GROUP UBISOFT

- Independent from productions
- R&D, common technologies
    - Destruction, Navmesh, UI, Networking, …
- Domain experts
- Mandates on productions

# REALBLAST

Realtime Destruction Solution

- A complete destruction solution:
  - Runtime destruction library
  - Fragmentation tools (3d creation sw)
  - Destruction properties editor (editor / 3d creation sw)
  - External debugger

# Collaborations

- Mandates on a few productions
- First shipped destruction
  with AC IV : Black Flag
- Now a core feature of TOM CLANCY'S RAINBOW SIX SIEGE

# Destruction in Rainbow Six: Siege


Attackers want to get in
Defenders block and funnel invaders
Destruction
TOM CLANCY'S
RAINBOW SIX | SIEGE

# Dynamic Environments in R6:S

- Destruction as a gameplay opportunity
  - Procedural destruction
    - Precision & endless possibilities
  - Reinforcement and barricades
    - Shape up the environment
- Opens us a lot of different strategies and ways to counter them

# Dynamic Environments in R6:S

Trapdoors

Barricades

LOS Floors

Breachable walls

# Sounds simple!

# Reality Check

- Visually coherent destructible environments come at a price:
  - Model “destruction-ready”
  - Destruction creep

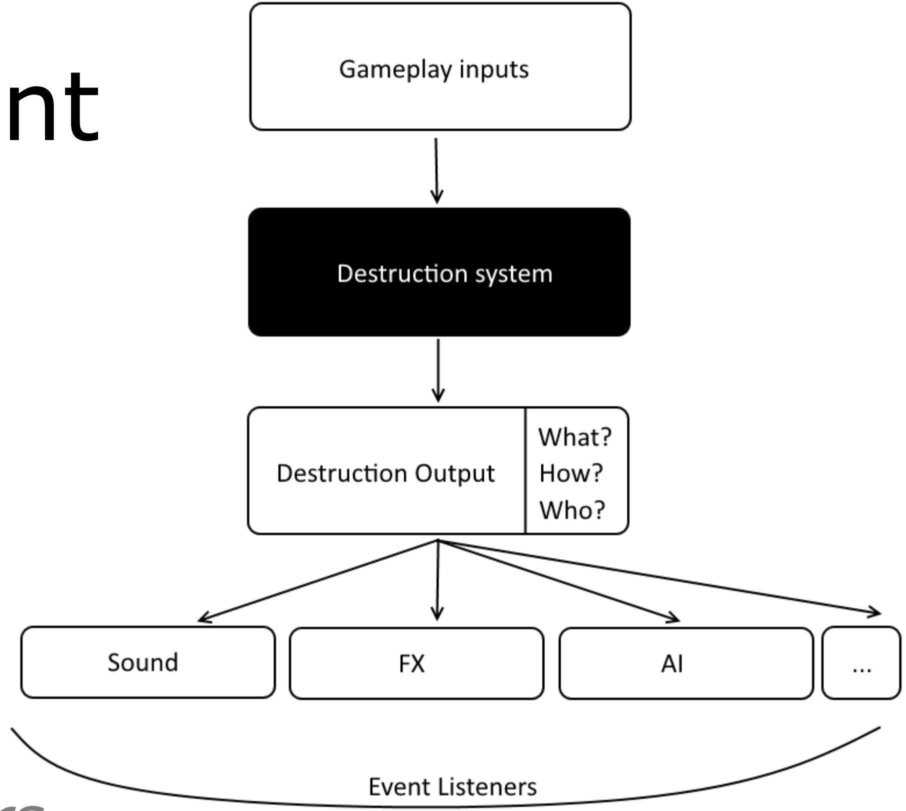
# Reality Check

- Artist training is needed
  - Changes modeling style and technical details
- Game changer for designers
  - Visual language, control, new variables
- Performance Cost
  - Strain on the engine (physics, rendering)
  - Runtime destruction

# The R6:S Game Ecosystem

*Designing for interaction with other subsystems*

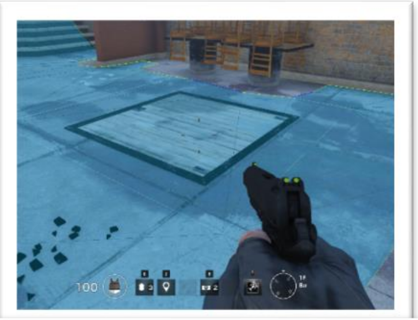
# Destruction Event System




*Hint: some care needed to have asynchronous listeners*

# AI Navigation: Navlink Update

Trapdoors



Breachable Walls

# AI Visibility / Sound Propagation

- Destruction changes the acoustic of the environment drastically
    - Sound (& propagation) is an important feature of R6:S!
- AI visibility through partially broken walls

# Gameplay Elements

- Need to have “state-like” behavior on top of procedural destruction
- Need to know when an object is broken
  - Ambiguous concept
  - Can be managed through properties (static vs dynamic) or state (triggers)

# RealBlast Tech

*Concept, model and procedural destruction*

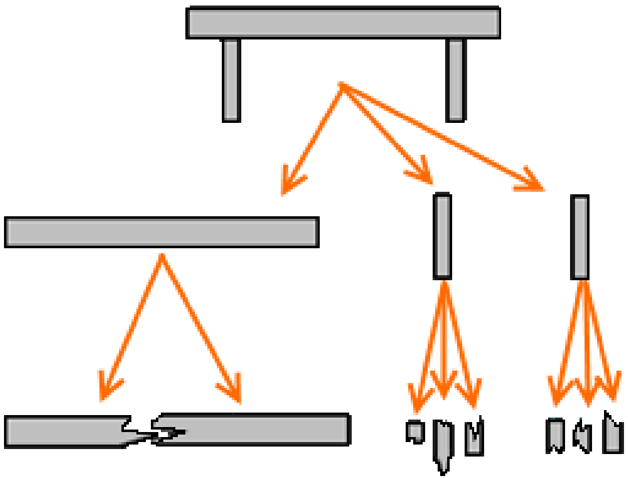
# Destruction Model Concept

- Objects are separated into different parts based on their physical material.

*Hint: should drive modeling for assets to be more readily destruction compatible*

# Destruction Model

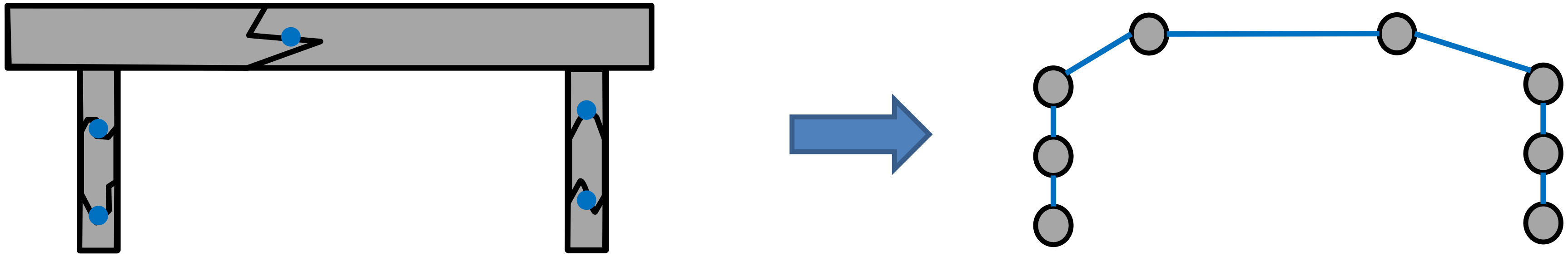
- Hierarchical decomposition
  - Based on fragmentation



*Hint: efficient for rendering & physics*

# Destruction Model

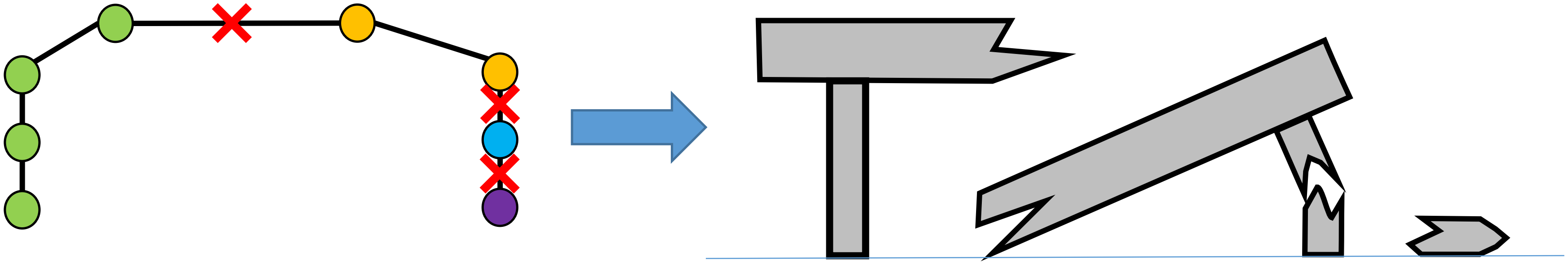
- Connection-based leaf graph



*Hint: simple concept, algorithms known*

# Destruction Model

- Game interacts with connections
- Leaf graph manages state



# Proc. Integration in the Model

- Leaf fragments can be flagged as procedural, depending on topology

- Visual and collision can change
- Can create new child fragments
- Create connections from parent’s

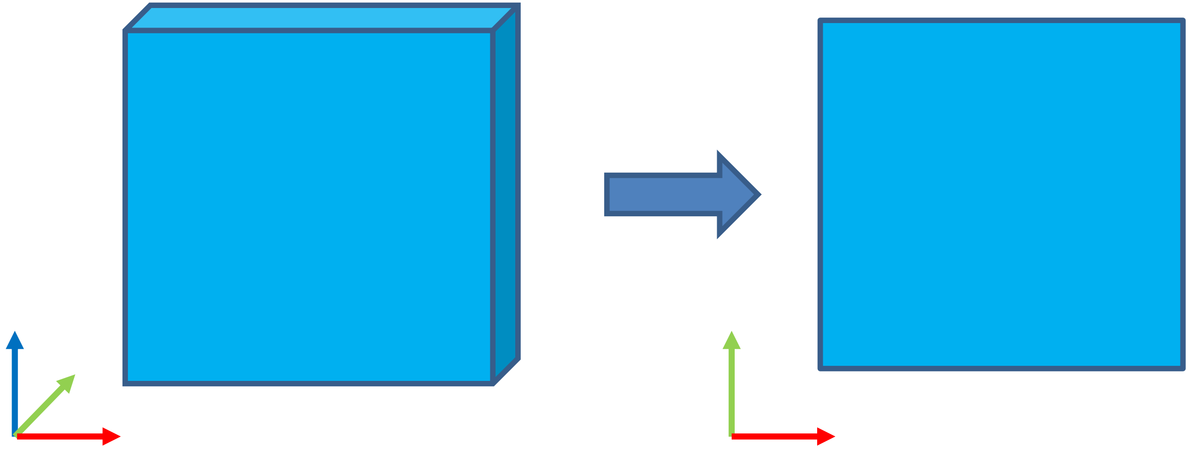
# Surface Procedural Destruction overview

# Surface Procedural Destruction

- Developed exclusively for Rainbow 6: Siege
- Use arbitrary cutting polygons to cut a planar surface
  - Great flexibility & simplicity to implement cutters
- General 2D polygonal technique
  - **Robust, fast, simple**

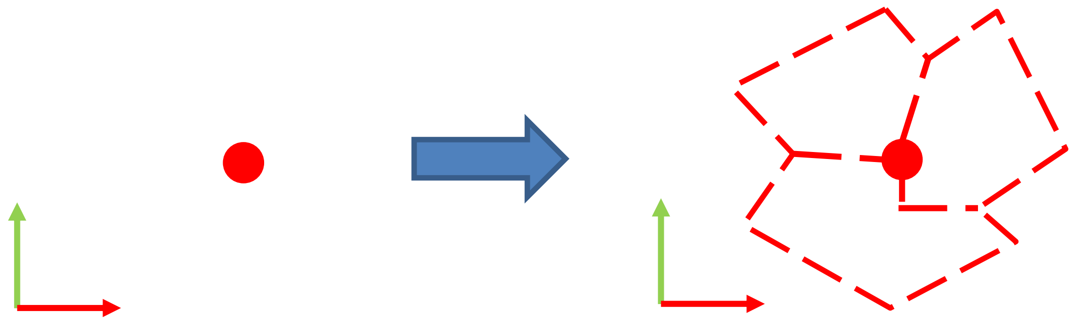
# Creating a 2D Model

- 3D -> 2D projection



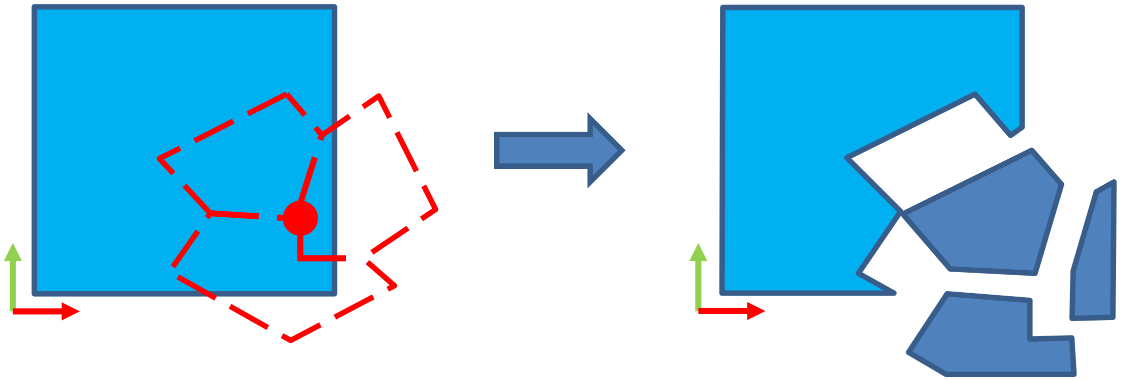
# Generating a Cut Pattern

- Shape depends on:
  - Impact position in local space of object
  - Combination of inputs and material parameters (cutter)



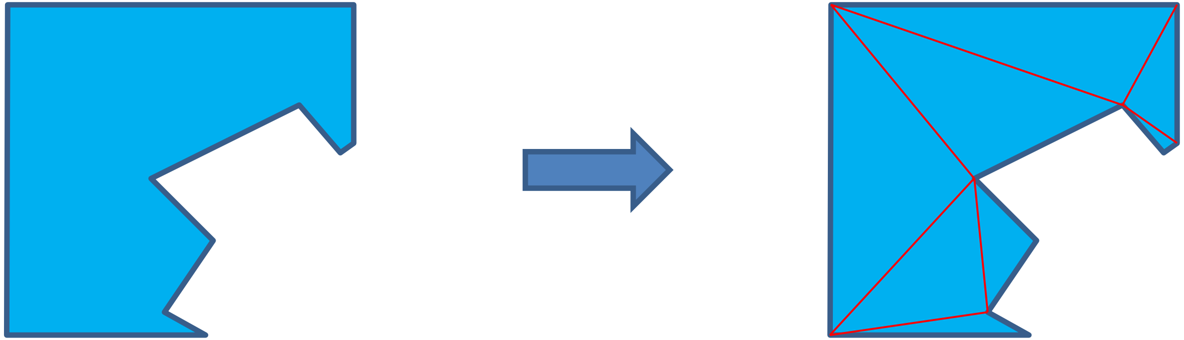
# Polygon Intersection

- Intersect surface polys vs. cutter polys
- Simple example: Weiler-Atherton polygon clipping algorithm



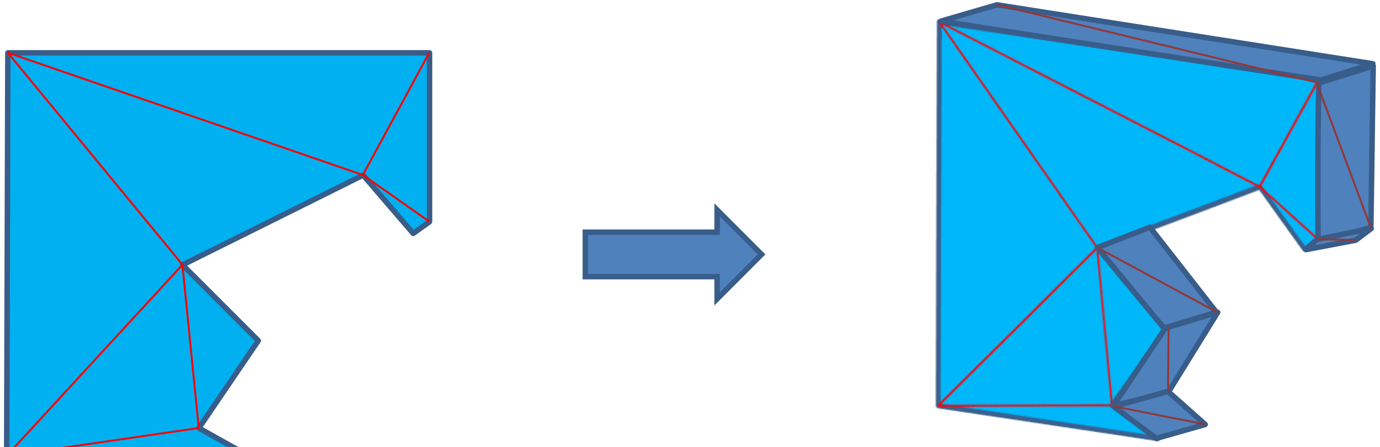
# Triangulation

- Ear Clipping:
  - Robust
  - Can handle multiple holes



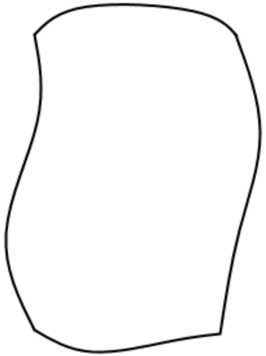
# Creating a 3D Model

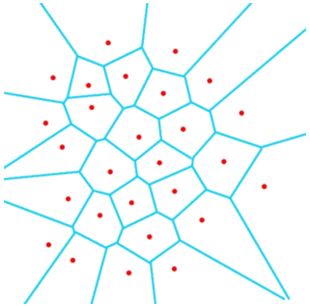
- Extruded 3D mesh from 2D surface

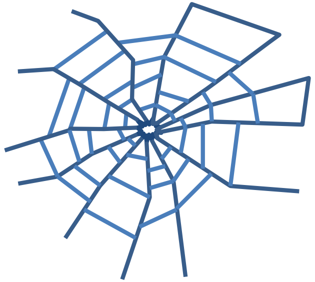


# Cutters

- Different classes of cutters
  - Some define a perimeter
    - Examples: Random ellipse, spline
  - Some define inner fragments
    - Ex.: Voronoi
  - Some define both
    - Glass, Texture

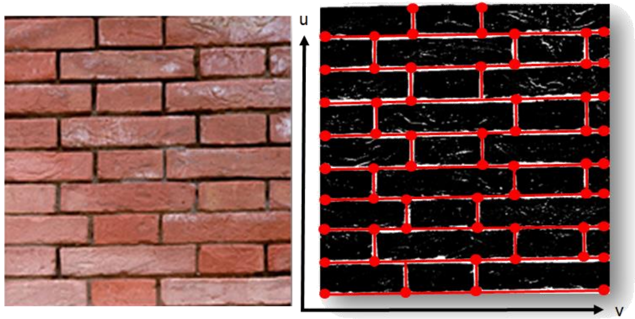






# Texture Cutter

- Continuous and tileable motif mapped in UV
  - Pattern is generated in UV space, then transformed to 2D surface space
  - Artists use a tool to generate vector coordinates



# So, what do we have now?


BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE

# Improving the Visual Look

- Traditionally done as decals on the GPU side
- Decorations VS Decals
  - Decorations output actual geometry

  \+ More flexible (*esp. for transparency*)

  \+ Preserved through destruction and child surfaces

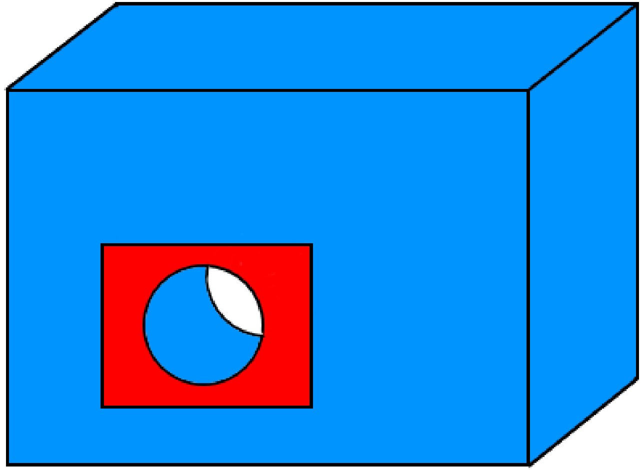
  \- *More costly for cpu/rendering/memory*

  \+ *Can be applied offline by artists (marks & imperfections) work*

  \- *More work*

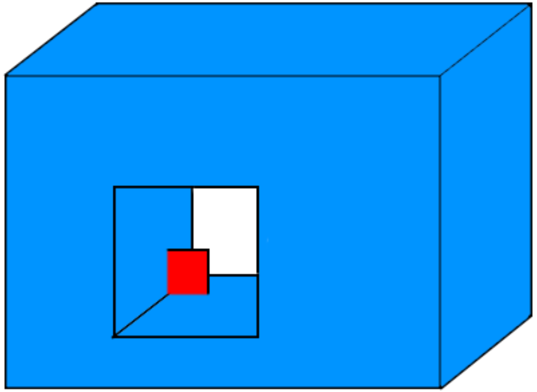
# Cut Decorations

- Decal-Like on surface:
  - Planar meshes that stick on the surface
  - Cut along with the surface



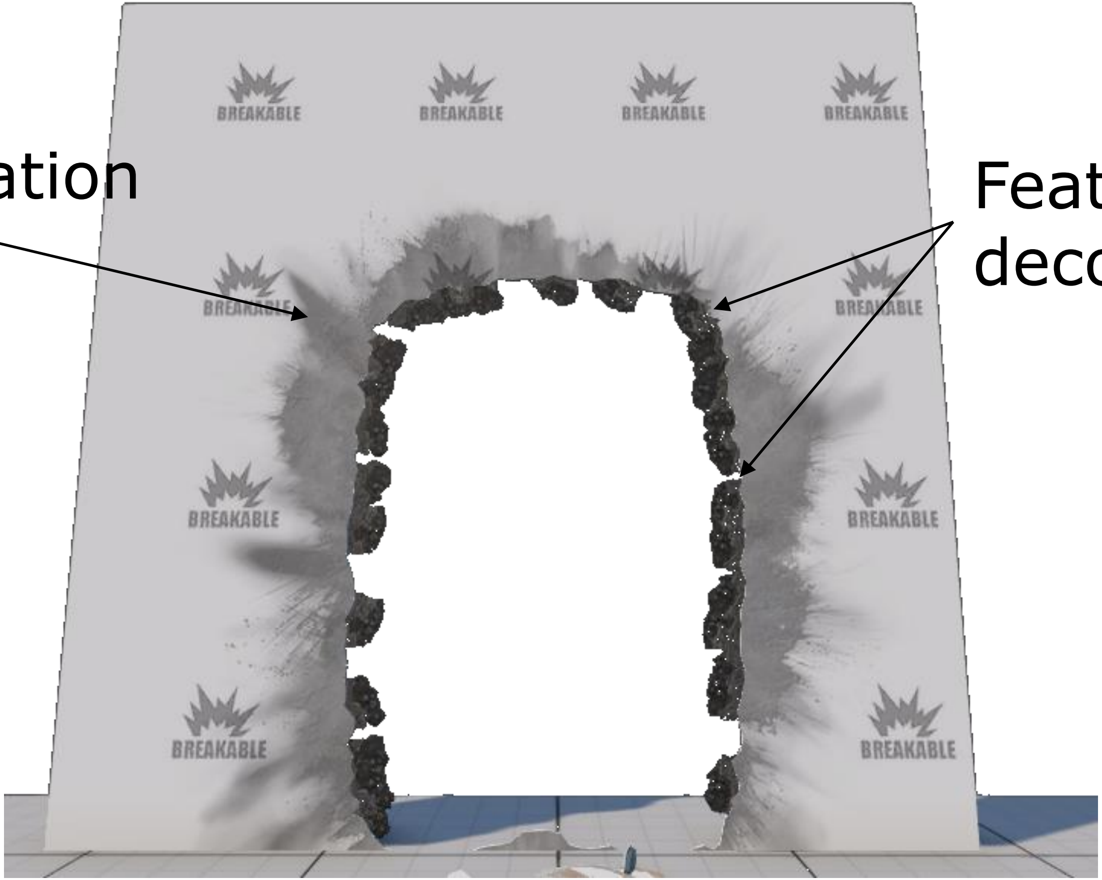
# Feature-Bound Decorations

- Decorations attached on geometric features
  - On edges and vertices
  - Not cut, just disappear when the feature is gone
  - Can protrude from the surface



Cut decoration

Feature-bound decorations



# Destruction & Performance

*Running at 60 FPS*

# Impact on Other Systems

- AI, AI navigation, sound propagation need to be more dynamic
- Rendering:
  - Static lighting and shadows are severely limited
  - Less occluders, can see more objects
  - **See Jalal's talk: "Rendering Rainbow Six: Siege"**
    **TODAY 3:30 in Room 2006, West Hall**

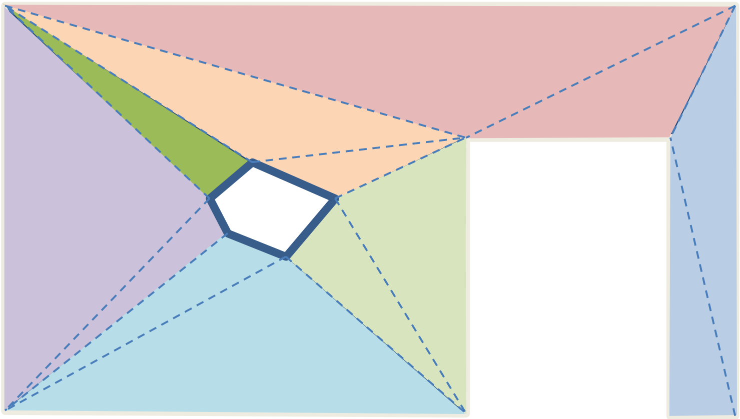
# Collision Update

- After destruction, very likely concave
- Use visual for collision?

*Hint: efficient use of the physics layer (feed planes instead of convex) is a win*

# Collision Update: Our Solution

- Collection of 2D convex shapes from a simplified version of the surface
  - Remove small holes
  - Reduce tessellation of holes



*Hint: we use the actual surface geometry for hi-resolution collision (e.g. shooting)*

# Debris in R6:S

- Performance choice:
    - **No** procedurally cut dynamic fragments
    - Well-placed replacements
        - Instanced
        - Recycled aggressively

# Debris in R6:S

- Other tricks:
  - Dynamic fragments don't collide together
  - Vaporize fragments on explosion
  - Simple collision primitives (*always boxes*)
  - Havok FX

# Destruction Performance

# Making it Real(time)

- The initial requirements were simple:
  - **Performance** : 60 fps for smooth gameplay
    - Destruction should not deteriorate framerate
  - **Determinism**:
    - Every player should experience the game the same way, i.e. all gameplay-related elements need to behave the same for all players

# Destruction Budgets

- CPU:
  - Pre-fragmented: **not a risk**
    - We had shipped AC:IV before, and still optimized it.
  - Procedural: **high risk**
    - Given roughly 6ms for a wall (2 procedural layers + pre-fragmented)
- GPU memory: 25 MB
- RAM: 200 MB data + 150 MB engine

# Memory

- Not a huge issue on the destruction side
  - Kept our footprint as lean as possible
    - Strip off parts not needed in working-data resources
    - Lean down usage-based data (ex.: vertices, segments triangles)

# Performance

The **most risky** part of destruction

- Very data-driven -> **hard to contain**
- **Punctual**

-> Use a collection of techniques and ideas

# Multithreading

- Multithreading at the object-basis is trivial
- Each independent sub state is MT in the simulation
- MT procedural destruction
  - *Didn't go to MT algorithms, but might be a next step*

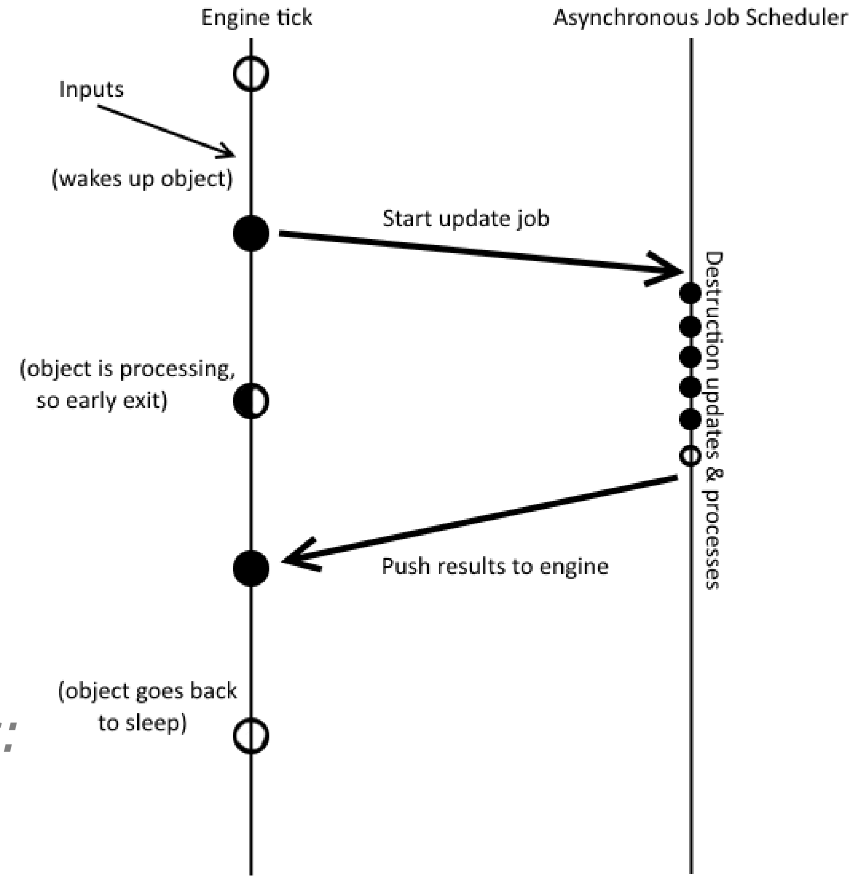
*Hint: watch out for race conditions when creating new data*

# Asynchronicity

- Made destruction a manageable risk
  - Little impact on framerate and game feel

- Introduces delay in game perception vs. actual destruction state.
  - Creates the need for an event forwarding mechanism

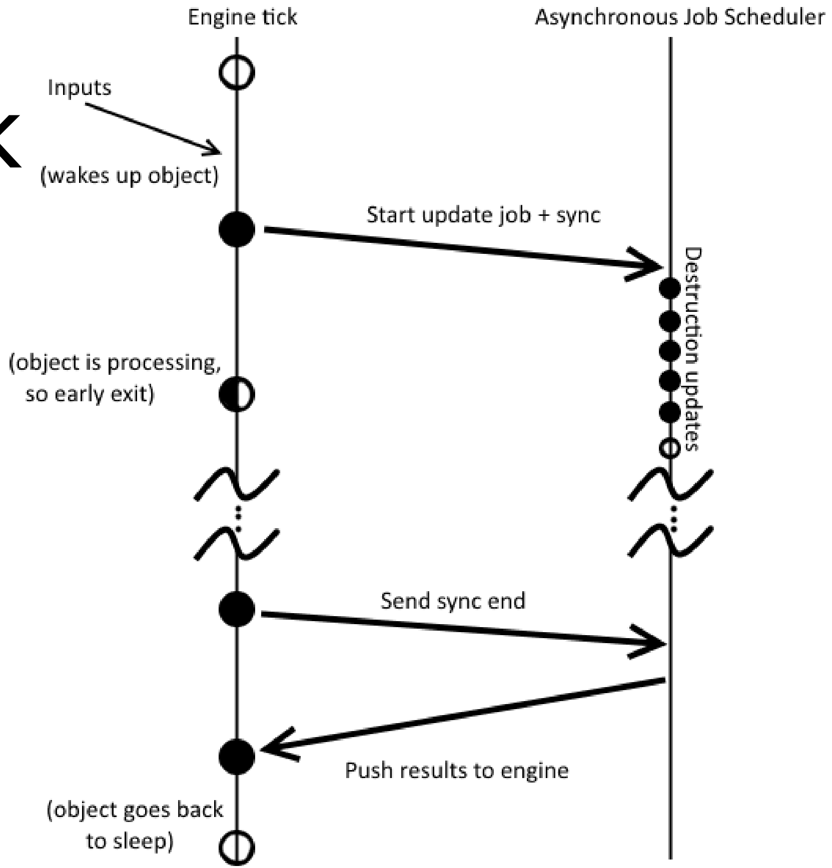
# Asynchro-How?




*Hint: caveat on the asynch scheduler: you might not want to run along with physics*

# Asynchronicity Trick

- Enables **Pre-destruction**
  - Perform destruction in advance
  - Synchronize with end of animation



# Time Slicing

- Asynchronicity not sufficient to be engine-friendly
  - What about a 60ms spike?

- Optimize or time-slice to prevent these issues

# Time Slicing

- Simple time-slicing on the destruction side:
    - Functions are split into steps
    - Not finished? Rescheduled

- No easy solution in C++*
- State variables can double-up as working data (e.g. preallocated resources)

# Slicing in Practice (simplified)

- Somewhat intrusive macros, but can easily be disabled

```
#define START_STEP_FUNCTION( stateVar, state ) bool fellThrough = false; \
                                   switch(stateVar) \
                                   { \
                                   case state: { \

#define STEP_FUNCTION( state ) }; break; \
                                   case state: {

#define END_STEP_FUNCTION( exitVar) }; break; \
                              default: fellThrough = true; } \
                              if(!fellThrough) return exitVar;
```

```
bool StepFunction( int & state )
{
   START_STEP_FUNCTION(state, 0)
      A();
      ++state;
   STEP_FUNCTION(1)
      B();
      ++state;
   STEP_FUNCTION(2)
      C();
      ++state;
   END_STEP_FUNCTION(false)

   return true;
}
```

# Time Slicing, or not

- Multi-threaded, time-sliced code is very hard to follow and debug
  - Make sure you can disable it easily
  - If possible, make it single-threaded as well!

- While this may hide some problems, it will make debugging tractable 95% of the time.

# Optimization

- Code/Algorithms optimization is a given

- But,
  - Measure Performance & Optimize
  - Limit Degenerate Cases
  - Keep Runtime Allocations Low

# Measure Performance & Optimize

- Without measurements, the best you can do is guesstimates and stabs in the dark
  - Captures from: Testers, Game sessions, Automatic tests
  - Interested?

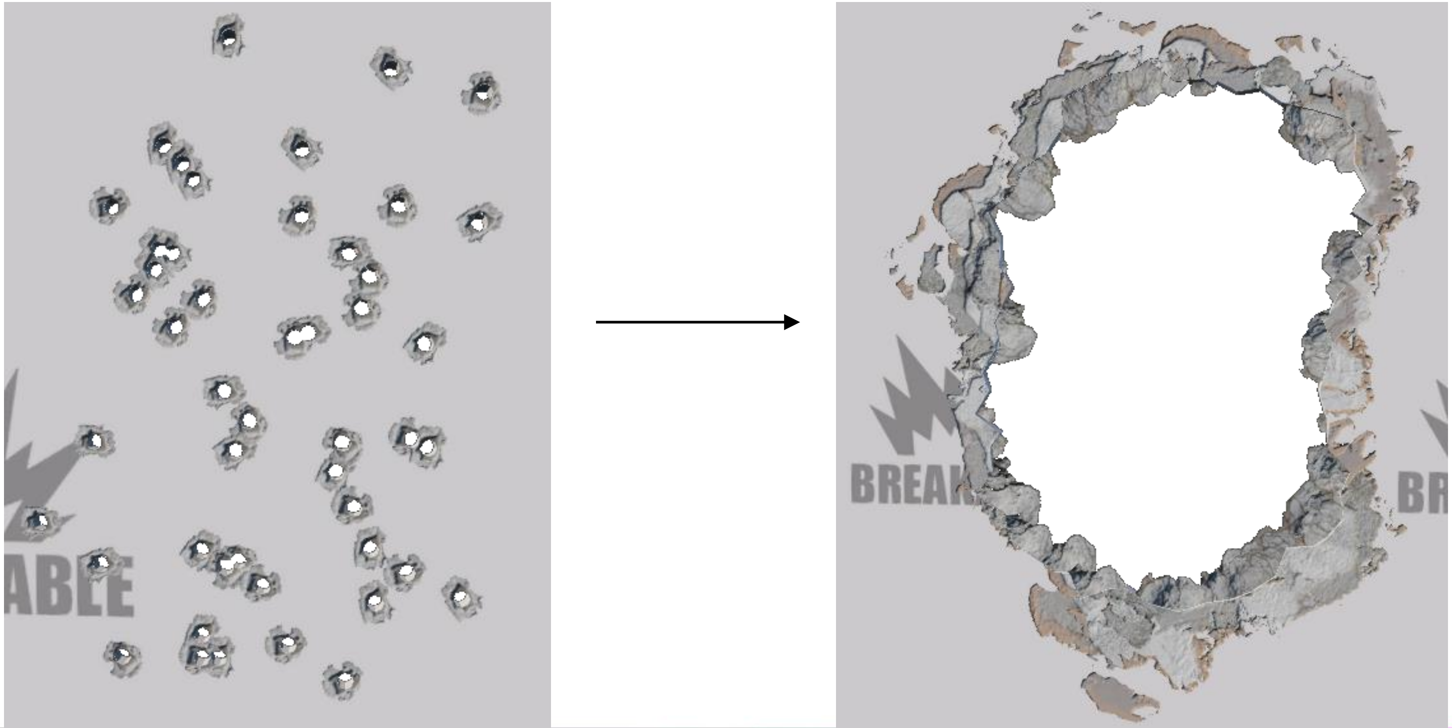
    ***See Unified Telemetry, Building an Infrastructure for Big Data in Games Development by M. De Pascale***

    ***Tomorrow at 5:30 PM West H. Room 2006***
- Optimize bottlenecks

# Limit Degenerate Cases

- Police somewhat the data
    - Hopefully through artist training
- Disable features that are not needed anymore
    - Example: dynamic fragments don't re-break from explosions
- Implement features to help bound complexity

# Self-Destruction



# Limit Runtime Allocations

- Avoid runtime allocations
  - Often locks in multithreaded environments
  - Use mostly for data that needs to be kept
- Strategies:
  - Hybrid heap/stack arrays
  - Working data structures
  - Pools of short-lived objects
  - In-Place algorithms

# Benchmarks

| | PC | PS4 | XB1 |
|---|---|---|---|
| Single bullet hole | .33ms/.36ms | 1.1ms/1.4ms | 1.1ms/1.5ms |
| Single explosion (drywall layer) | 1.4ms/1.9ms | 2.8ms/4ms | 3.6ms/4.9ms |
| Single explosion (2 drywall + 2 wood layers) | 8.1ms/10.3ms | 19.5ms/23.5ms | 19ms/23ms |

To take with a heap of salt

# Online

*Determinism and Replication*

# Determinism and Replication

- Gameplay feature -> **deterministic** and **replicated**.

- **Minimize** bandwidth usage over CPU usage
    - ✓Events (messages)
    - ✘States (meshes)

*Hint: easy to do JIP with events*

# Physics Replication, Debris & Determinism

- Physics replication is hard
  - Destruction is asynchronous
  - Characters impact dynamic objects

⇨ **Not replicated**

- All dynamic objects and debris are always small
  - Ignored by gameplay
  - Destruction on dynamic objects not replicated

# Determinism

- Contract between game and destruction

We expect to be provided:

- The **exact same inputs**
- In the **same order**

# Determinism Requirements

- Same inputs
  - Not trivial:
    - Race conditions between gameplay states
    - Network data compression even locally
    - Need symmetrical compression

$$\forall v, C(v) = c^{-1}\big(c(v)\big)\ \textit{where c is compression}$$

$$\rightarrow C(v) = C\big(C(v)\big)$$

# Determinism Requirements

- Same order
  - On an object-basis
  - On R6:S, guaranteed by the network layer
    - Which is definitely the easiest solution by far

- Still had to make the code not too sensitive to “same frame” vs. “different frames” events.

# Randomness vs. Determinism

- Seed a RNG based on some input value
  - On R6:S : based on impact position

    *Assumes perfect replication of inputs*
  - With enough granularity, the player will never see it's the same
- Store the RNG on TLS for ease-of-use
  - Caveat: time-slicing

# Different States vs. Determinism

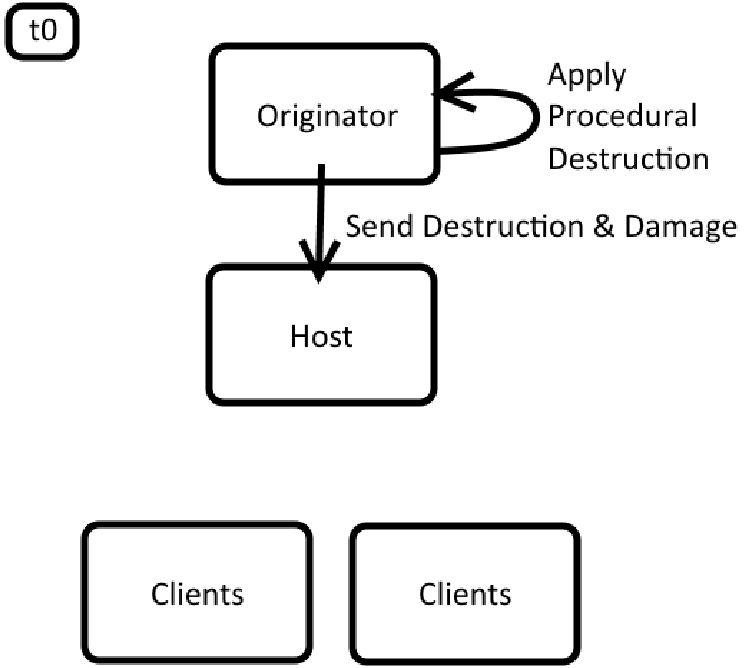
- Incoming events can affect elements in the past or in the future
  - Events happening on clients & hosts at the same time
  - Bursts of events
  - Asynchronicity

# Instant Feedback vs. Determinism

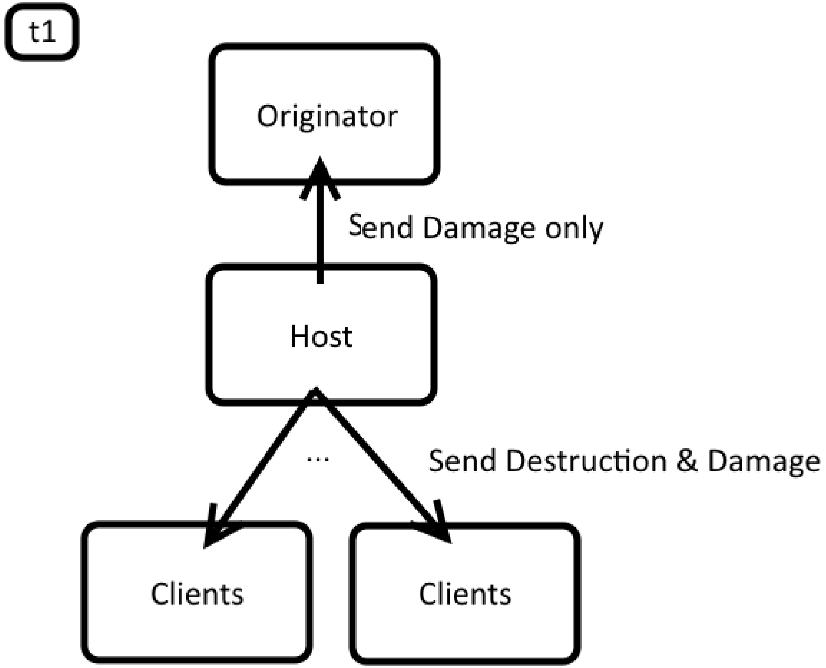
- Instant feedback for shooting in R6:S is needed
  - latency over the internet >> latency on LAN
  - Breaks contract (*ordering*) -> breaks determinism?

- Initially, **compromised** replication because of the self-destruction feature

# Instant Feedback vs. Determinism



# Instant Feedback vs. Determinism



# Caveats

- **Not** a **perfect** solution
  - Originator might not end up with **exactly** the same state
- In practice, the difference is **minimal** and unlikely to cause issues

# Other solutions

- The **rollback**
  - Each client:
    - Keeps track of locally applied events
    - Reverts and re-applies when receiving other events from the host
  - Pros/Cons:
    - ✓Super robust and deterministic
    - ✗Stack of events to revert is not really bounded (susceptible to latency)
    - ✗Each revert step is memory-intensive (full surface backup)

# Future Development & Conclusion

# Future Development

- Tools
- Curved surfaces *(already in S1)*
- New destruction types/behaviors
  - Plastic deformation
  - Stress analysis
- More (look, optimization, …)

# Takeaways

- Destruction **deserves** a dedicated team
  - Many fronts on the tech side (runtime, gameplay, tools)
  - Imposes restrictions and forces training on the production side
- Needs a clear **production buy-in**
  - A lot of teams need to contribute and adapt for dynamic environments

# Takeaways

- Must be **tackled early on**
  - R&D investment
  - Production & mentalities inertia

# Conclusion

- Dynamic environments are **here to stay**.
- Destruction is **awesome**.

So…

- Promote innovation and change
- Bite the bullet

# P.S.

- Jalal’s talk:
  - *Rendering ‘Rainbow Six: Siege’*

  *3:30 PM Today, Room 2006 West Hall*
- Maurizio’s talk:
  - *Unified Telemetry, Building an Infrastructure for Big Data in Games Development*

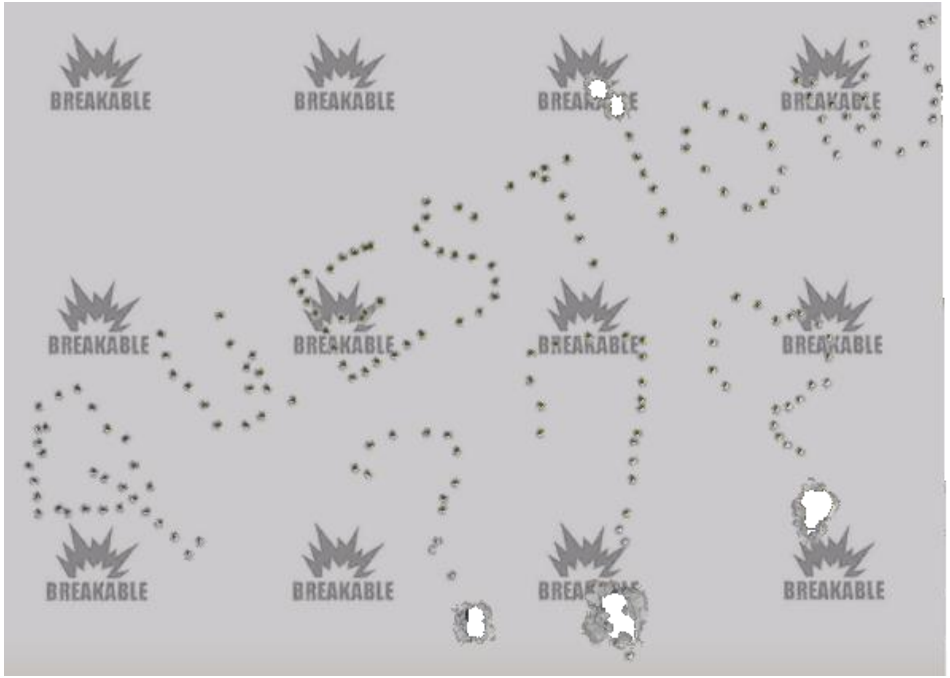
  *5:30 PM Thursday, Room 2006 West Hall*

# P.P.S.

Reach me at:

julien.lheureux@ubisoft.com

# Questions?


BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE
BREAKABLE

UBISOFT

# MORE QUESTIONS?

## MEET ME ON THE UBISOFT LOUNGE

TODAY from 4PM to 5PM

WEST HALL, 2ND FLOOR

@UBISOFTCAREERS #UBIGDC