



March 21-25, 2022
San Francisco, CA

Resurrecting a Classic: Bringing Diablo II into the 3rd Dimension

Kevin Todisco,
Lead Software Engineer, Graphics
Blizzard Entertainment

#GDC22



Hello! Thank you for joining me today. I'm Kevin Todisco, Graphics Engineer at Blizzard and I was the lead graphics engineer on Diablo II: Resurrected. I'm excited to have the opportunity today to present all the hard work of the team. Before I start I want to say don't forget to fill out those surveys at the end of the talk so that I can make my talks even better and, if I'm lucky, get invited back in the future.

Now, I've worked for almost 10 years at Vicarious Visions, which is now part of Blizzard, but if you know Vicarious...



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

then you know we've had our hands in a lot of popular remasters over the past several years, so you could say we know a thing or two about remasters.



[image source: IGN]

And remasters, they generally fit into one of two categories: either you're rebuilding the entire game from the ground up - a remake, new code, new assets, probably some changes to the game itself



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

or you're increasing the quality of all the assets and behavior in the original to meet modern technology - a remaster, same code, same assets but upscaled, increased resolution and framerate, game remains the same, and generally 2D-to-2D or 3D-to-3D.



I'm here today to talk about something a little different. I'm here to talk about how we remastered Diablo II while also adding a dimension to it. But it's still a remaster; it's not some PR spin when we say that all the original code is still running - all the original code is still there and running just like it did in the year 2000, and that's what is controlling the game. But, now we have these cool new 3D graphics to represent it.



As we can see in this concept art, our Art team had some high ambitions. They wanted to create very high fidelity assets for a super realistic and gritty reproduction of the classic, dark Diablo aesthetic. Diablo's graphics in the year 2000 may certainly look dated today but back then they were really convincing. A player could let their mind's eye fill in all the gory detail and paint a picture of what they were getting immersed in. For this remaster, we had to capture that.



As we can see in this concept art, our Art team had some high ambitions. They wanted to create very high fidelity assets for a super realistic and gritty reproduction of the classic, dark Diablo aesthetic. Diablo's graphics in the year 2000 may certainly look dated today but back then they were really convincing. A player could let their mind's eye fill in all the gory detail and paint a picture of what they were getting immersed in. For this remaster, we had to capture that.



As we can see in this concept art, our Art team had some high ambitions. They wanted to create very high fidelity assets for a super realistic and gritty reproduction of the classic, dark Diablo aesthetic. Diablo's graphics in the year 2000 may certainly look dated today but back then they were really convincing. A player could let their mind's eye fill in all the gory detail and paint a picture of what they were getting immersed in. For this remaster, we had to capture that.



As we can see in this concept art, our Art team had some high ambitions. They wanted to create very high fidelity assets for a super realistic and gritty reproduction of the classic, dark Diablo aesthetic. Diablo's graphics in the year 2000 may certainly look dated today but back then they were really convincing. A player could let their mind's eye fill in all the gory detail and paint a picture of what they were getting immersed in. For this remaster, we had to capture that.



On the other hand, we made an explicit choice to reuse the engine, in large part because we did not want to risk the chances of getting any intricacy about the game wrong. So as an engineering team, we had to solve the largest question of them all – how do we make this possible? The simulation is in 2D – as in, positions and logic are all conducted in screen coordinates or an arbitrary 2D game coordinate space. And we must build a pipeline for our team to make 3D assets, without touching the existing engine.

What are we talking about?

- What this talk is:
 - The engineering story of the project.
 - Light on code, heavy on engineering solutions.
- What this talk is not:
 - Why did we choose to use the old engine?
 - ***Remastering a Classic: 'Diablo II: Resurrected'***
 - Robert Gallerani
 - Room 2010, West Hall
 - Friday, March 25th
 - 3:00pm – 4:00pm

This is the story of how we solved this huge challenge with clever engineering and collaboration. This talk isn't so much a deep dive into the code, but rather how we engineered solutions to make the project possible.

What this talk is not is *why* we decided to preserve the original engine and build the project from that. If you're interested in the design motivation behind that, I definitely recommend checking out my colleague Rob's talk this Friday at 3 in Room 2010 in the West Hall.

What are our goals?

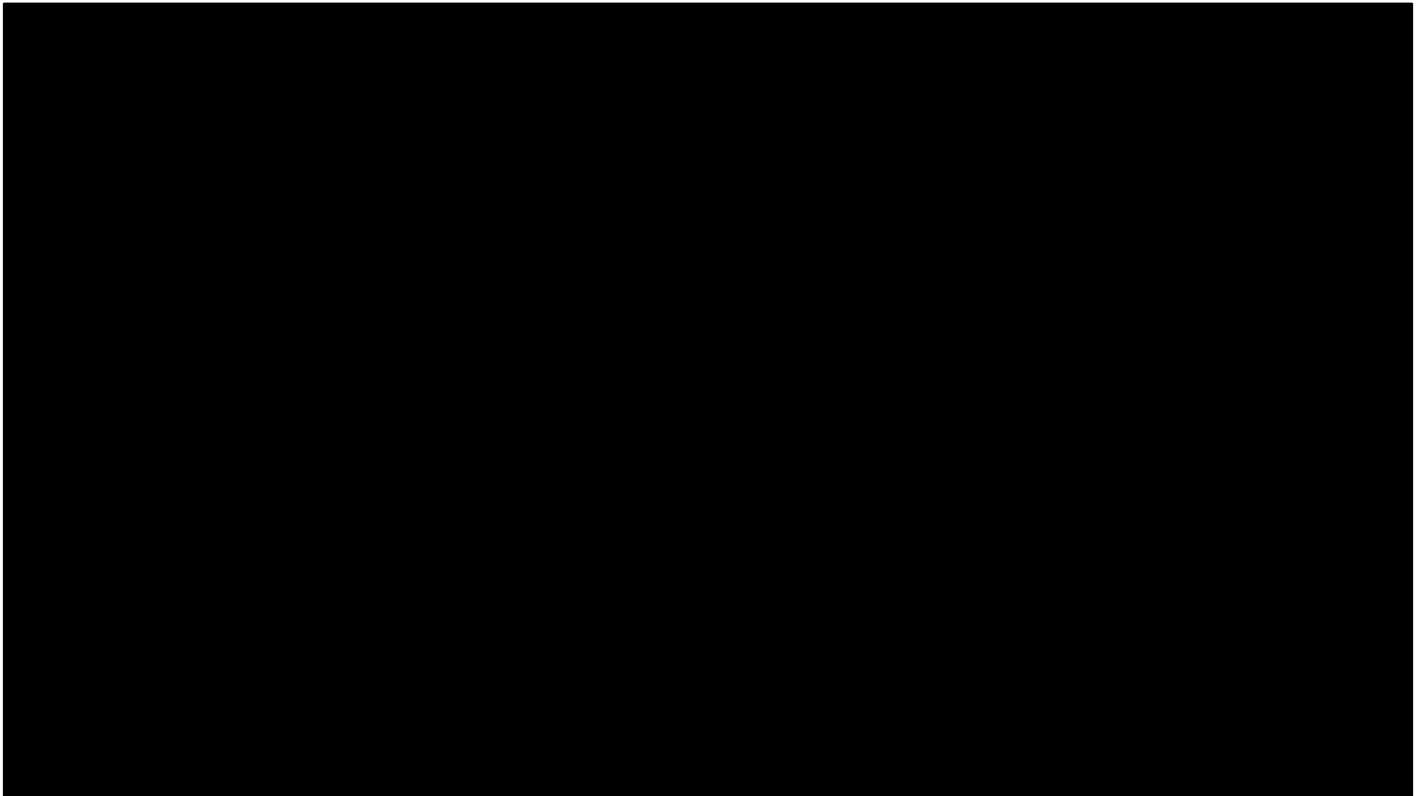
- In order to succeed, what technical challenges are there?
- Make the illusion of 3D real.
 - We're going to make a real 3D game.
 - How do we add an extra dimension to the game?
- Make it look like everyone remembered.
 - Art plays a big role in this, but so does engineering, as we'll see.
 - Visuals must service iconic gameplay.

In order to succeed here, we need to consider what technical challenges need to be overcome. First, that illusion of a 3-dimensional space that the original game created with its isometric camera? We need to make that real. We need to build out a 3D engine and figure out how to drive it from a 2D dimensional space. And second, we need to make the game look like everyone remembered. Obviously, art plays the biggest role in this but as we'll see, graphics technology plays a key role here as well.

How do we meet those goals?

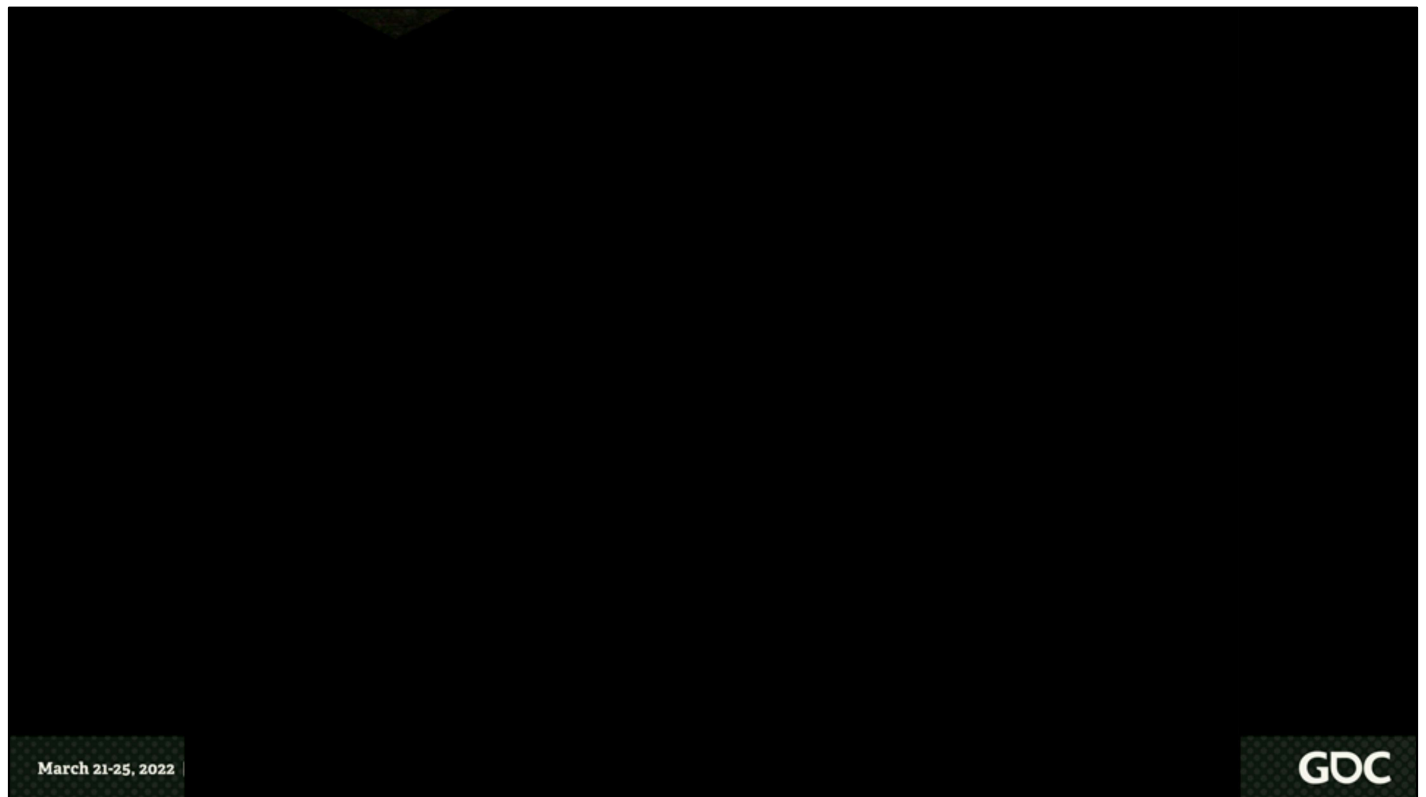
- We need a 3D engine.
- We need a toolchain.
- We need the right rendering technology.

And what are our steps to success? We need a 3D engine. We need a toolchain for art to be able to make the game. And we need to choose the right rendering technology that will enable us to create visuals that recreate what you saw in your minds eye all those years ago. So today I will walk you all through how we pulled this off, and the best place to start is in the year 2000, so... stay a while, and listen.



And how do we go about pulling this off? Three easy steps, right? We need a 3D engine. We need a toolchain for art to be able to make the game. And we need to choose the right rendering technology that will enable us to create visuals that recreate what you saw in your minds eye all those years ago. So today I will walk you all through how we pulled this off, and the best place to start is in the year 2000, so... stay a while, and listen.



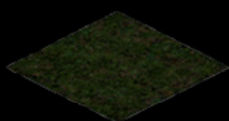


Diablo II was originally rendered using a collection of sprites created from pre-rendered 3D objects, split up into multiple pieces and then pieced back together to create various animations, procedural variability, and a variety of different level layouts.

Rendering begins with the floor, followed by shadows which are just sprites drawn at half height and skewed in a solid shade. Next up walls and units, such as buildings, characters and props are all drawn over top of the terrain. Then weather, and finally UI.

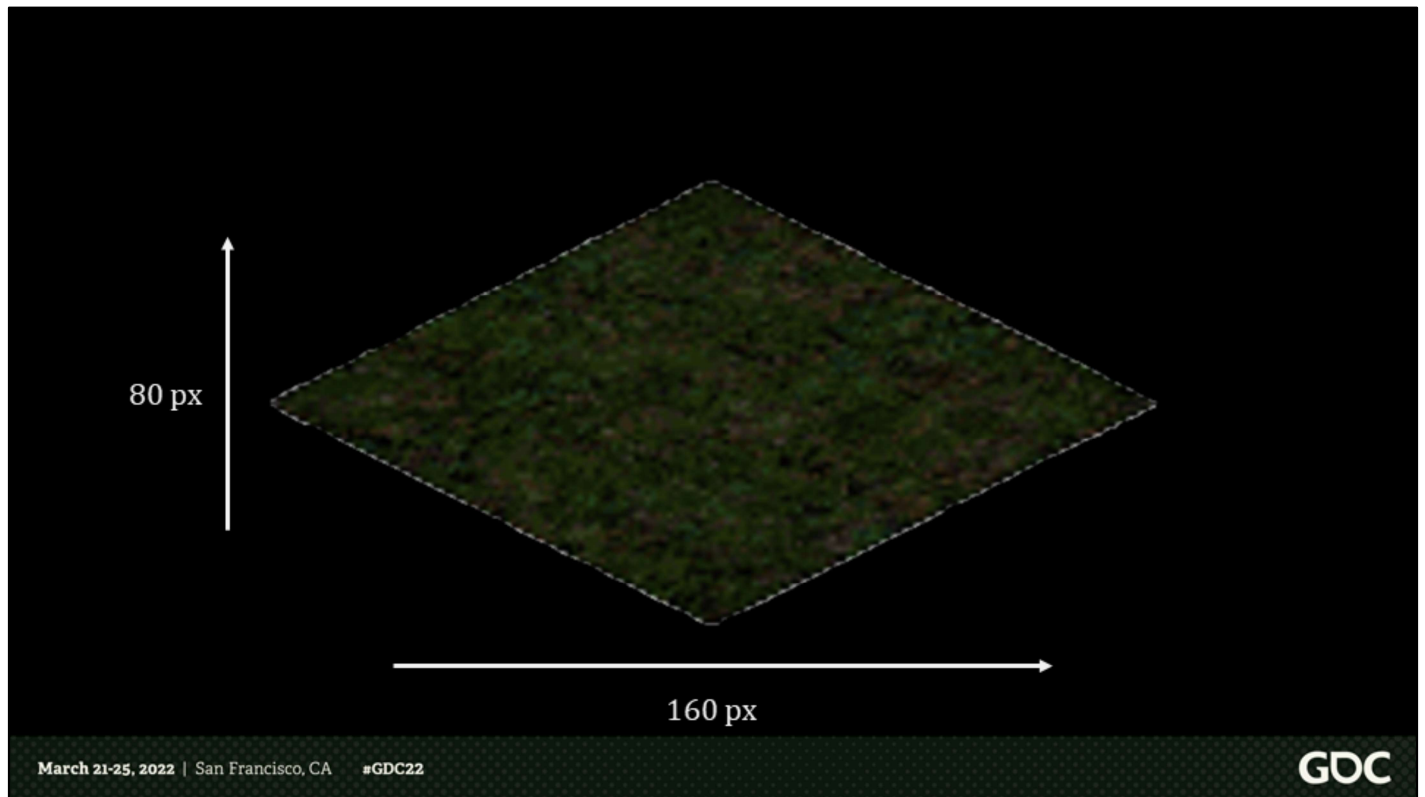


If we want to create a 3D space to sit on top of this world, we need to look at the original game as if it were actually 3D. One of the best clues for understanding this is a floor tile.

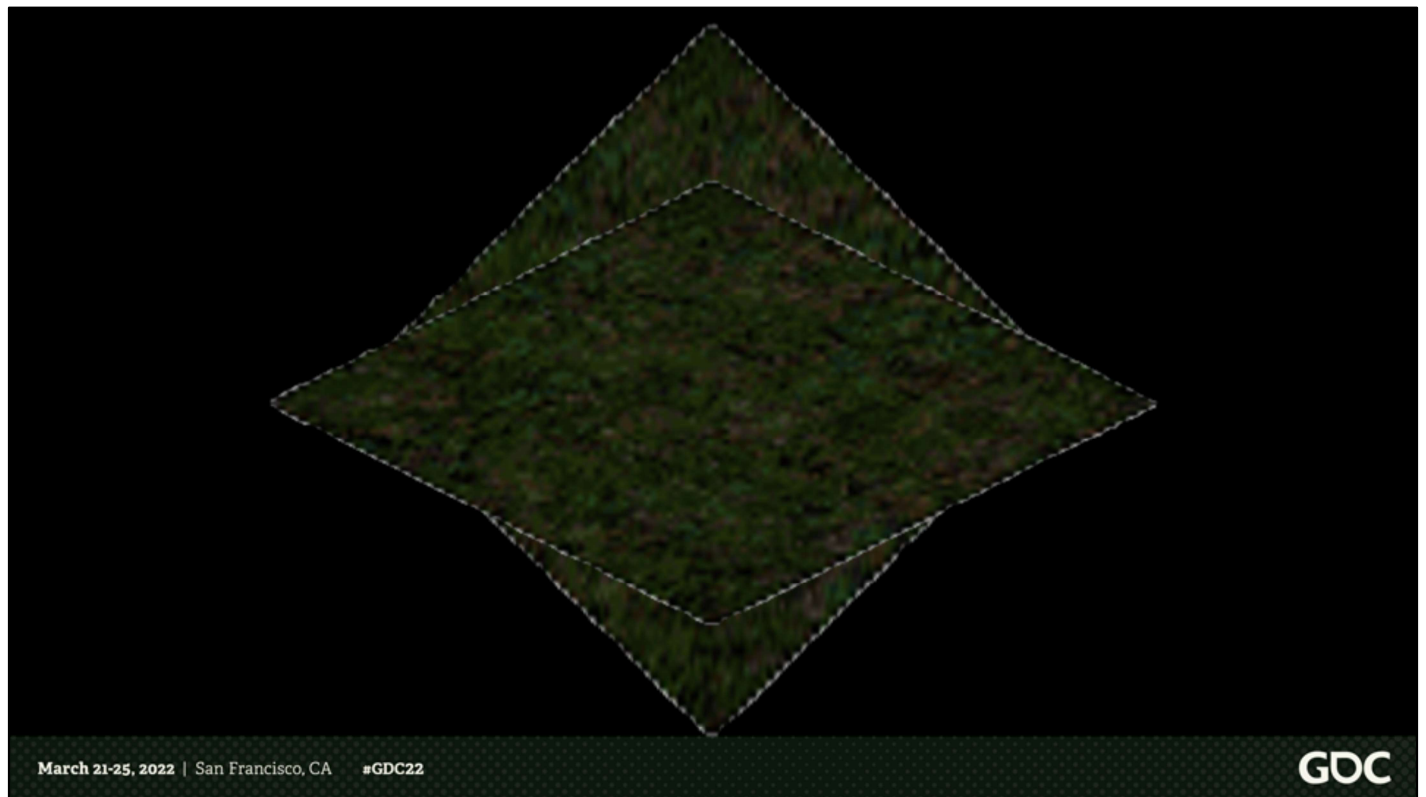


March 21-25, 2022 | San Francisco, CA #GDC22

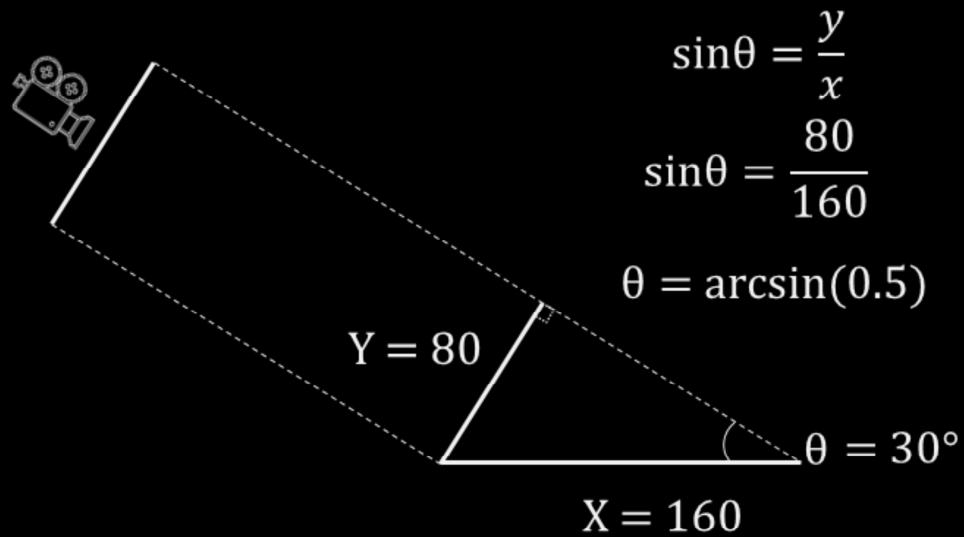
GDC



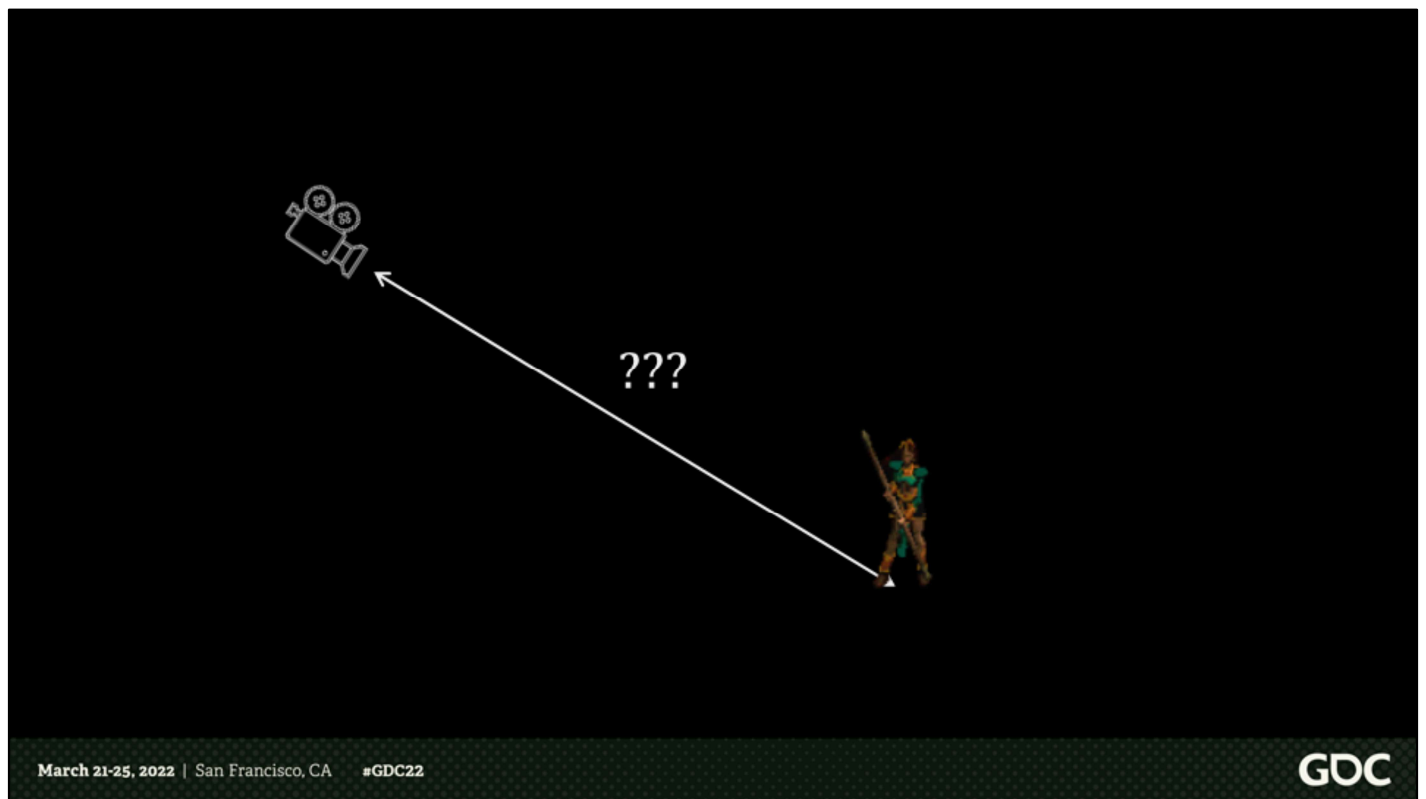
Floor tiles in Diablo II are half as tall as they are wide -- 80 pixels tall and 160 pixels wide. Assuming that, in 3D space, this floor tile is meant to be a square and sits on the XZ plane in the world, these dimensions give us all the information we need to calculate the viewing angle from which the tile is being looked at.



We start at the assumption that the tile in our mock 3D world is a square. This would mean that the tile has been foreshortened along the screen Y-axis by half of its diagonal length.



That foreshortening can be thought of as a right triangle formed by the projection of the tile onto the camera's projection plane. The tile itself is the hypotenuse. Its foreshortened diagonal axis is the short side of the right triangle, and the angle we need to solve for is the arcsine of the ratio of these two sides. Some quick trigonometry tells us that the camera is tilted downward 30 degrees.



The next question is, where is the camera located in space?



Floor tiles to the rescue again! Diablo II was originally presented at a resolution of 640x480, and the Lord of Destruction expansion -- ahem -- expanded that resolution to 800x600. Not only did it increase the resolution, but because it's a sprite-based game, it increased the viewable play space rather than just increasing pixel density. In either case, this means that a certain number of floor tiles can fit in one screen from left to right, and from this we can calculate the field of view of the camera.

Oh, I forgot to mention before now, Diablo II: Resurrected uses feet as its unit of measurement. Why? Because Diablo II used yards. Strange, I know. But as far as in-game units were concerned, a yard is 48 pixels horizontally and 24 pixels vertically in screen space, making floor tiles each $3\frac{1}{3}$ yards wide.

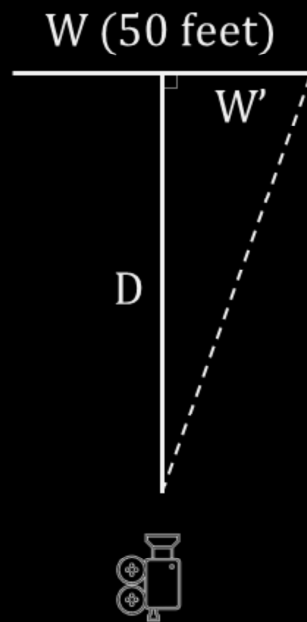


That makes the viewable ground space in this scene 16 and two thirds yards from left to right,



or 50 feet. From a birds-eye view, our camera will sit at some point in space looking at the game scene and will need to view 50 feet of space from left to right.

- 35mm film
- 400mm focal length
- "Squint test" to match perspective
- 0.16° FOV



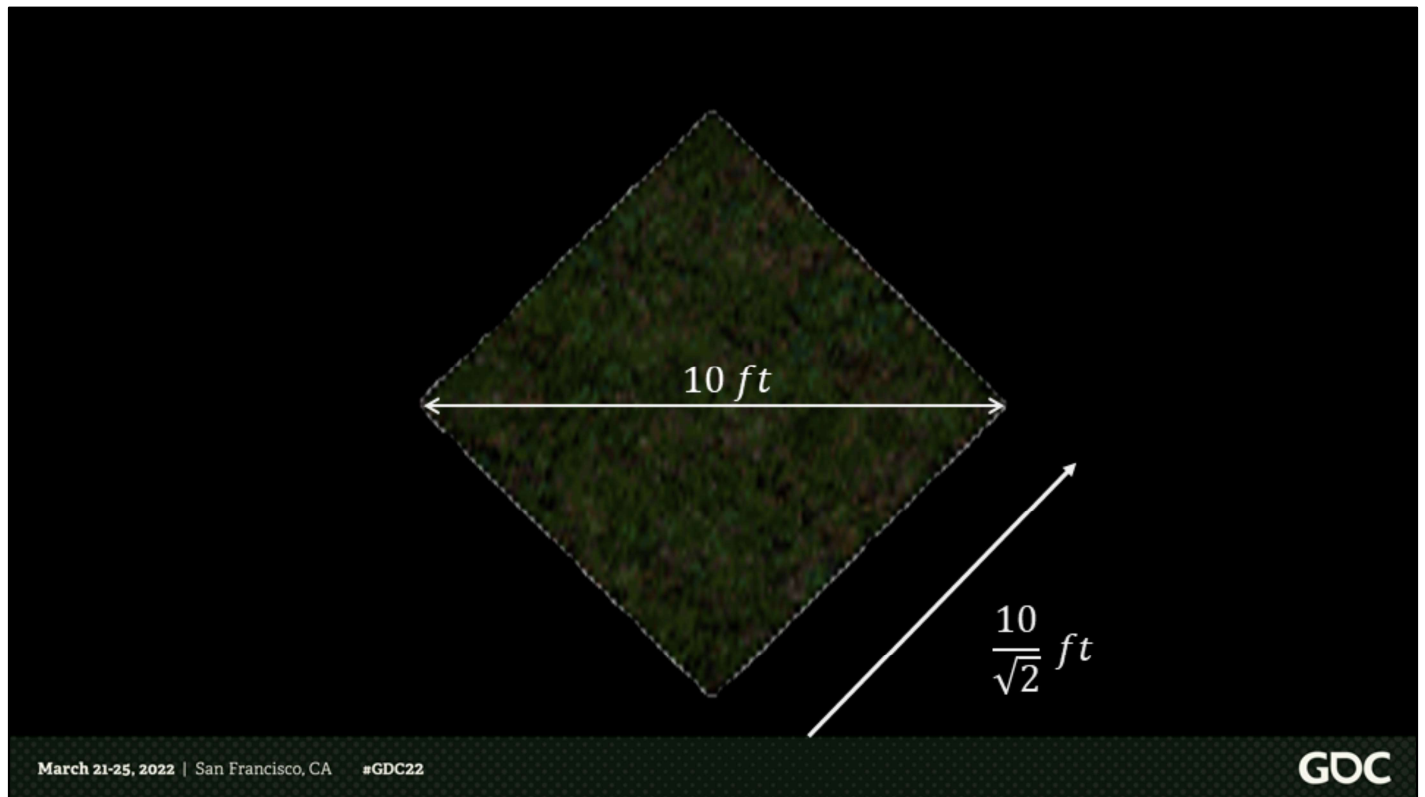
$$\sin \theta = \frac{W'}{D}$$

$$\sin(0.08^\circ) = \frac{25}{D}$$

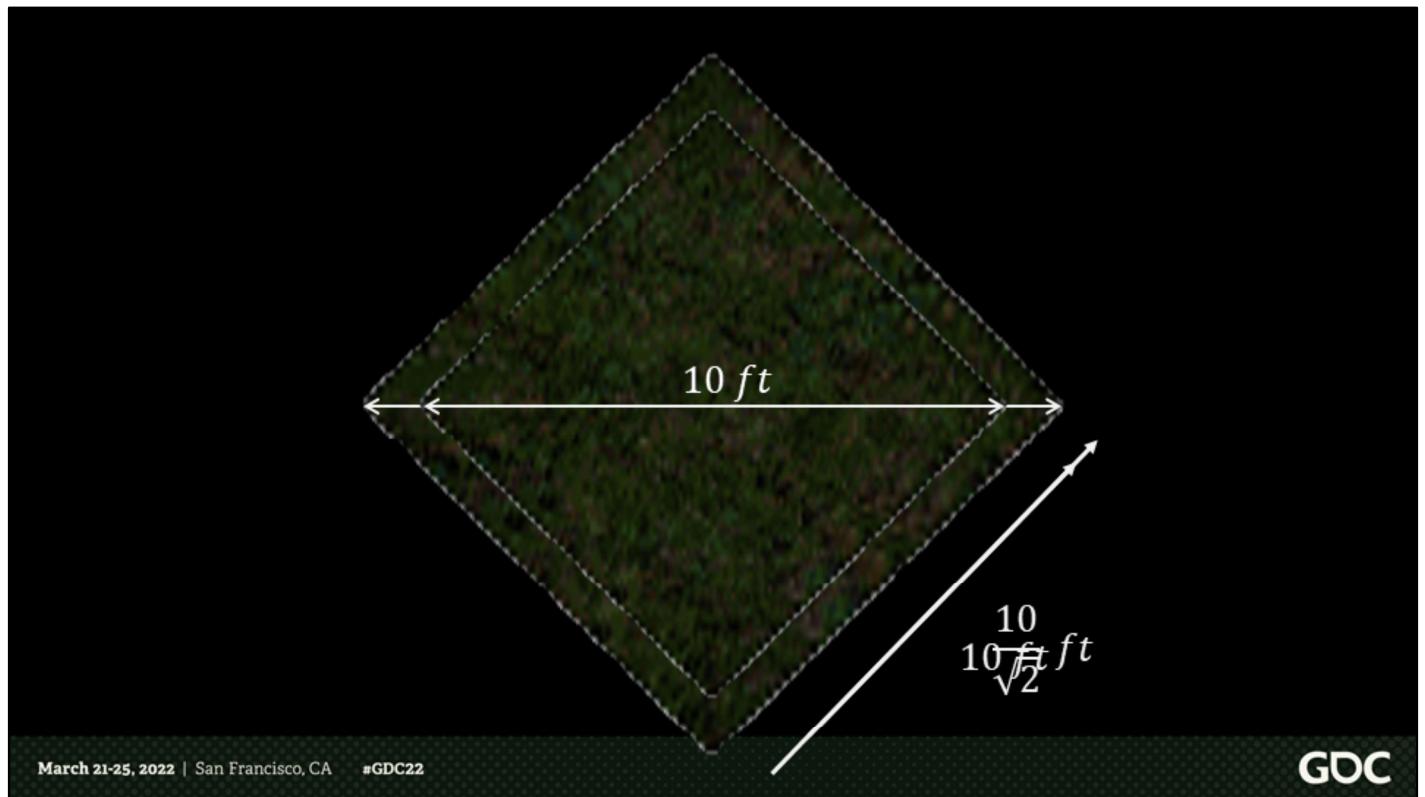
$$D = \frac{25}{\sin(0.08^\circ)}$$

$$D = 883.88$$

Assuming a standard 35mm film projection, which is 24mm in height, we calculated a 400mm focal length to match, using the squint test, the "fake perspective" mode that was introduced in the Lord of Destruction expansion, and we could use that focal length for the orthographic projection as well. The focal length can be converted to a field of view value of 0.16 degrees. Half of that gives our angle to our right triangle pictured here, and our W' value is 25 feet; rearranging we get our formula; and solving we get a distance from the focal point of 883.88 feet.



We also now know how large our floor tiles will need to be in 3D - 10 feet diagonally since five would fit across the screen in the 800x600 view, making up 50 feet of horizontal space. That means that the side of a floor tile would be 10 over the square root of 2 feet long, which is obviously not a very round number.



So in practice, we scale everything in the world up by the square-root of two, so that instead of being 10 feet diagonally, floor tiles could be 10 feet by 10 feet square. The reason for this is that 10 is exactly representable by floating point, and so we wouldn't have to worry about floating point imprecision creating issues at seams between the rooms that are procedurally stitched together to create the game world.

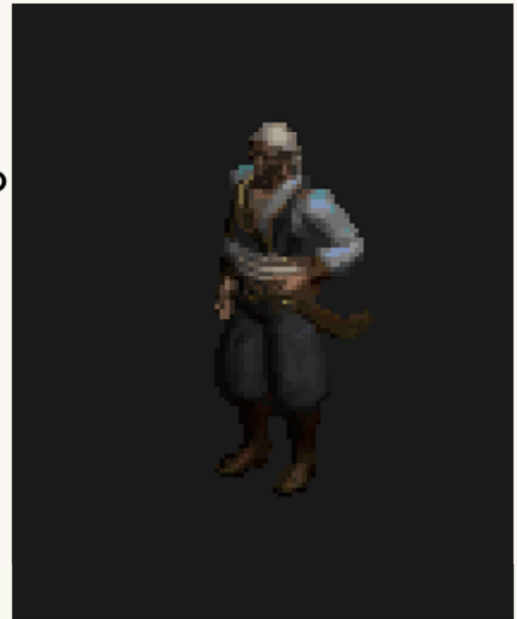
Orientation

- 64 different directions
- Treat as increments of 360°

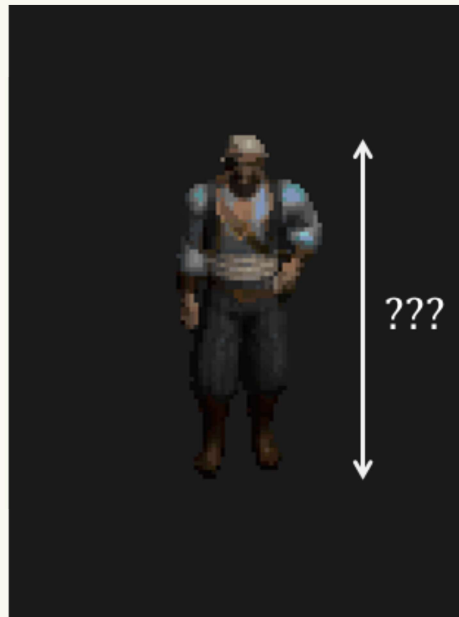
$$scale = -\left(\frac{2\pi}{64}\right)$$

$$offset = \frac{\pi}{4}$$

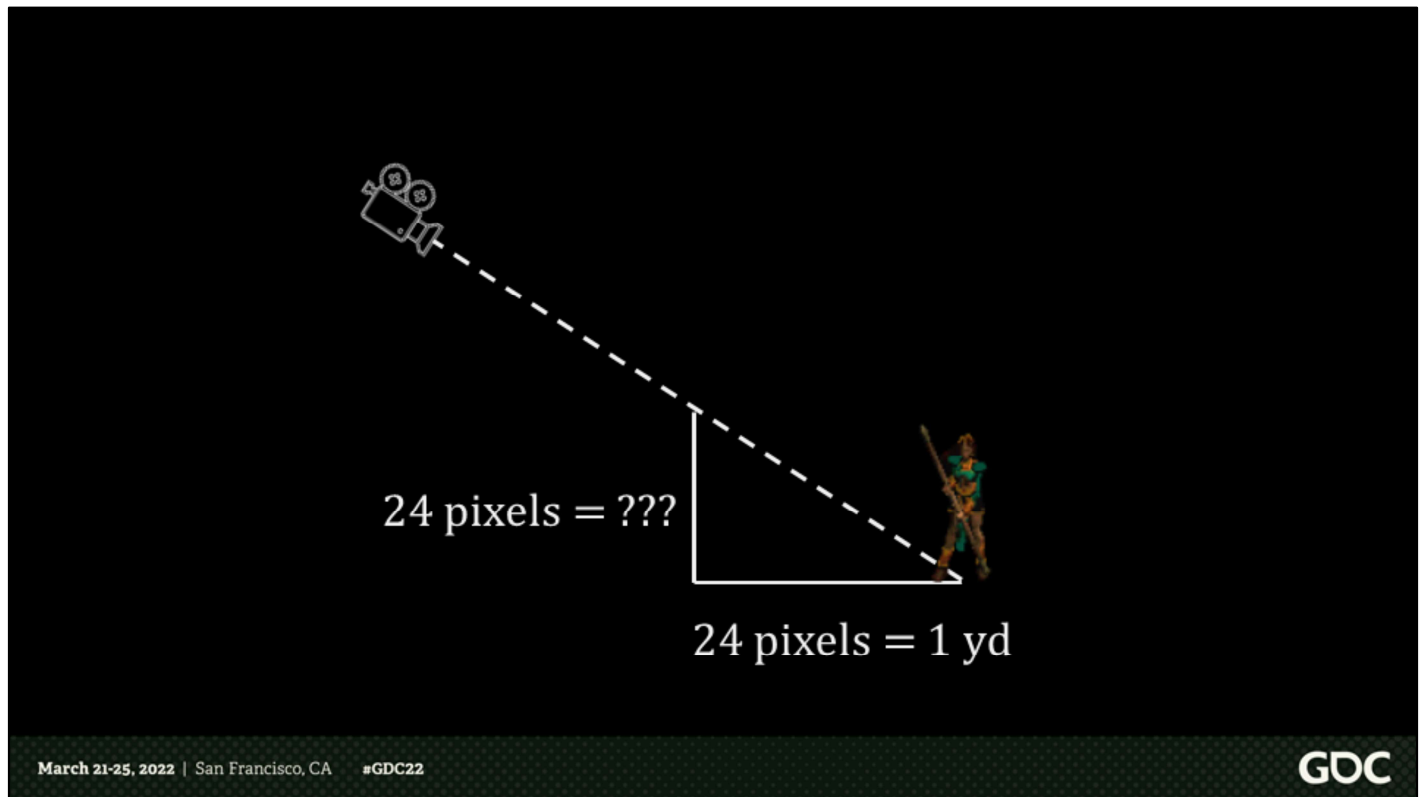
- "Neutral" is different in each space



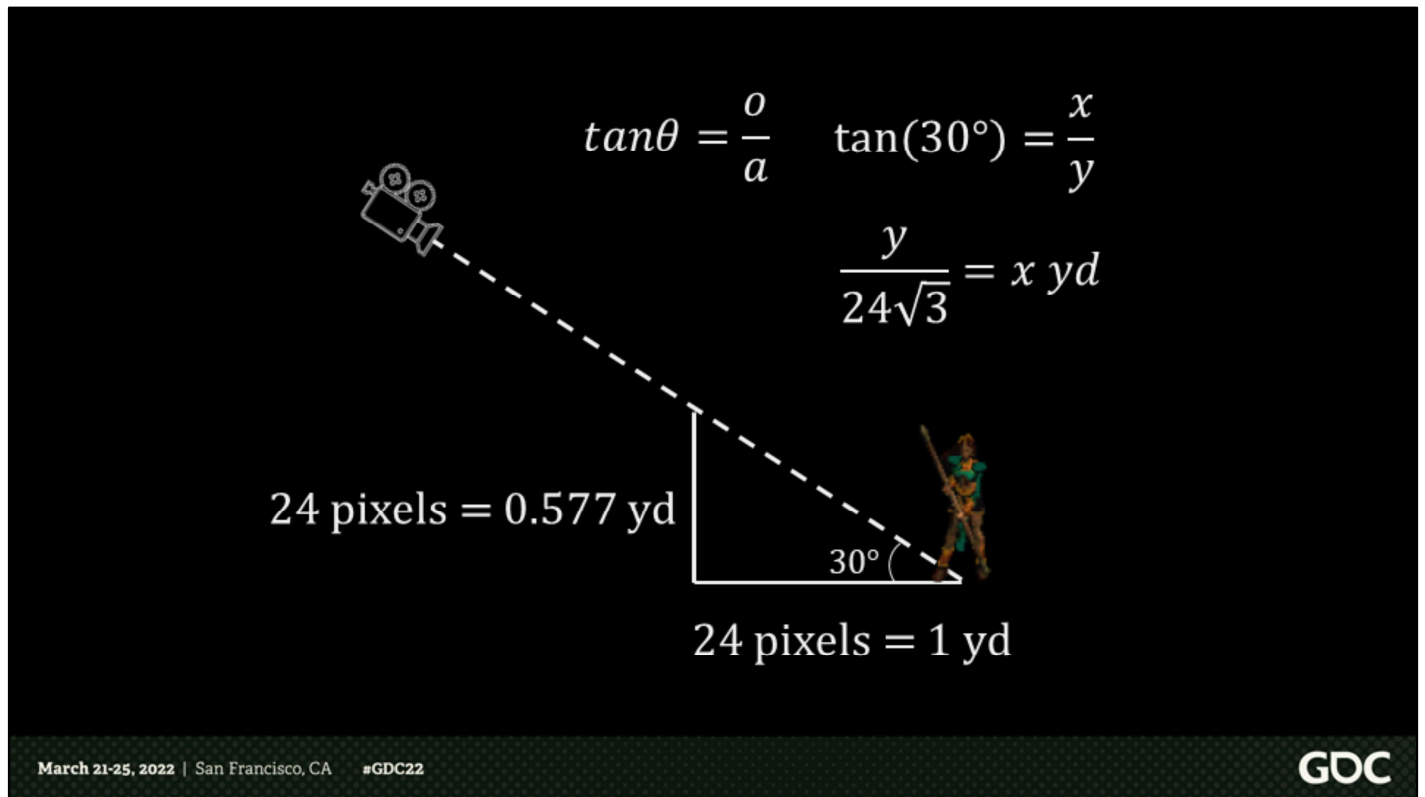
Another important question is how to orient everything? The original game allowed for a unit to have up to 64 different directions that it could face. We can treat these as 64 intervals of a 360 degree rotation. To convert from a 2D orientation value to a 3D rotation, we scale the value by the number of degrees in each increment, which is 2π over 64. It's negated because we have a right hand coordinate space. And, lastly we need an offset to account for the fact that "neutral" rotation in the 2D space is rotated 45 degrees from our forward axis in 3D space.



There's one more thing we need to figure out, and that's how to derive vertical height. Things like character sizes, missiles, and as we'll see soon, height changes in terrain, need this.

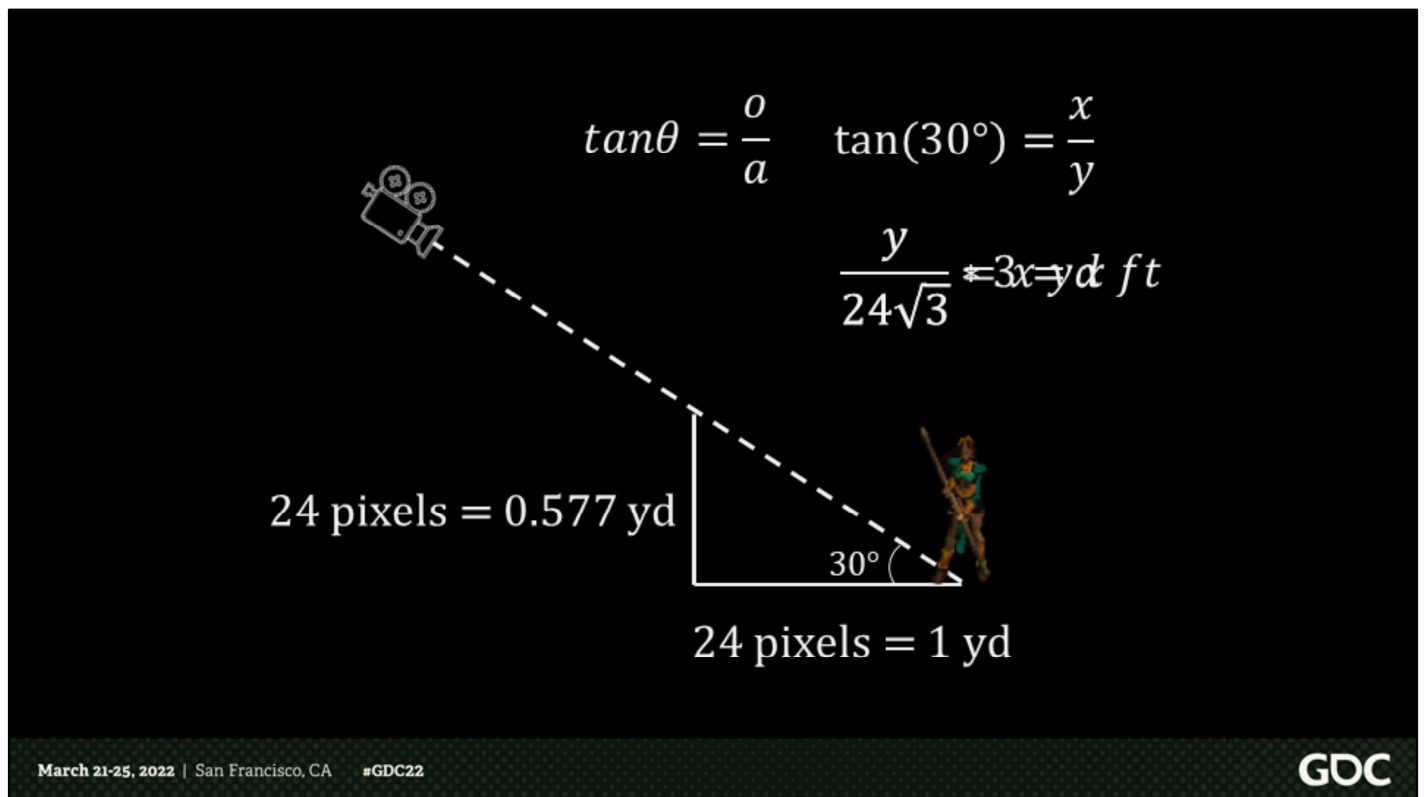


For calculating verticality from pixel measurements, we return to thinking about the 2D game as a 3D world. Pixels in vertical space along the screen could refer to two different directions, each of which has a different scalable value. Along the ground plane, 24 pixels in vertical screen space is one yard, or 8 pixels per foot. For height, it's something different. But once again, we can form a right triangle to derive the equivalent in vertical space.



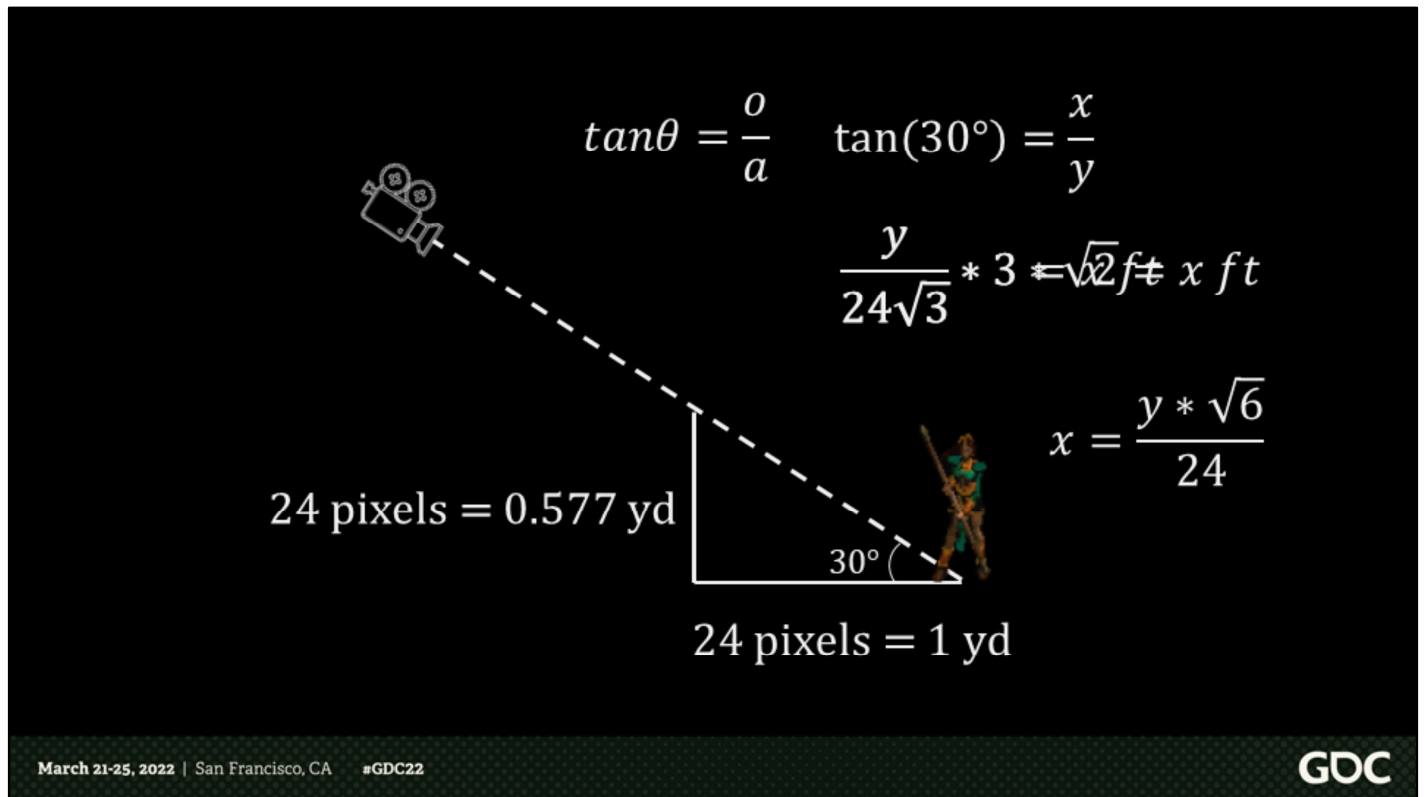
In this case our unsolved side of the triangle is parallel to the world y-axis rather than the projection plane, so we're dealing with the tangent of 30 degrees instead of the sine. The tangent of 30 is one over the square root of three, which means the same length projected onto the screen vertically is 0.577 yards, and the formula can be rearranged to show that the height in yards is the height in pixels, y , of the object on the screen, divided by 24 (at 24 pixels per yard along the ground plane) times the square root of three. That gives us height in yards, but we want feet as our in-game unit, so we multiply that by three, and then we finally multiply it again by square root of two as our world scaling factor.

In practice, we simplify this to a multiplication by the square root of six and division by 24. I'll leave it as an exercise to the audience to derive to that point.



In this case our unsolved side of the triangle is parallel to the world y-axis rather than the projection plane, so we're dealing with the tangent of 30 degrees instead of the sine. The tangent of 30 is one over the square root of three, which means the same length projected onto the screen vertically is 0.577 yards, and the formula can be rearranged to show that the height in yards is the height in pixels, y , of the object on the screen, divided by 24 (at 24 pixels per yard along the ground plane) times the square root of three. That gives us height in yards, but we want feet as our in-game unit, so we multiply that by three, and then we finally multiply it again by square root of two as our world scaling factor.

In practice, we simplify this to a multiplication by the square root of six and division by 24. I'll leave it as an exercise to the audience to derive to that point.



In this case our unsolved side of the triangle is parallel to the world y-axis rather than the projection plane, so we're dealing with the tangent of 30 degrees instead of the sine. The tangent of 30 is one over the square root of three, which means the same length projected onto the screen vertically is 0.577 yards, and the formula can be rearranged to show that the height in yards is the height in pixels, y , of the object on the screen, divided by 24 (at 24 pixels per yard along the ground plane) times the square root of three. That gives us height in yards, but we want feet as our in-game unit, so we multiply that by three, and then we finally multiply it again by square root of two as our world scaling factor.

In practice, we simplify this to a multiplication by the square root of six and division by 24. I'll leave it as an exercise to the audience to derive to that point.

Checkpoint

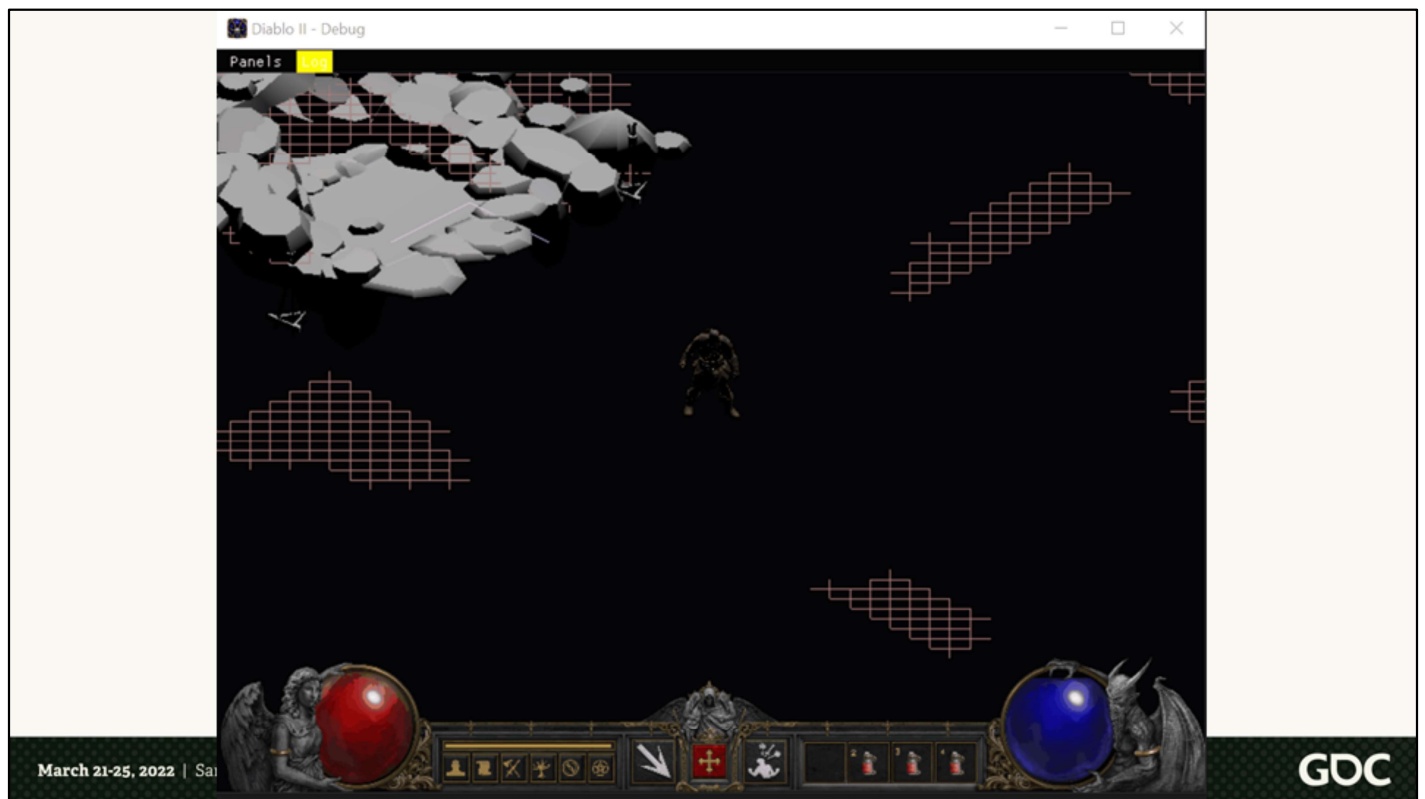
- We need a 3D engine.
 - We have a 3D space defined.
- We need a toolchain.
- We need the right rendering technology.
- How do we position things in space?

Ok, phew. We've reversed engineered the 2-dimensional space and worked out the mathematics of an equivalent 3-dimensional space. Now we have to figure out how we are going to position things in this world.

Converting Between Spaces

- Naïvely, treat the 2D game as existing on a flat plane.
 - 2D coordinates become coordinates on the XZ plane.
 - Spoiler, this isn't going to hold up later.

Initially, we can imagine that the whole 2D game is taking place on a flat plane. We've just imagined it as if it were 3D, so it's not a stretch to think that the whole thing can take place on a single plane. And so, we could consider the coordinates to be coordinates on the XZ plane in 3 dimensions. Will this work? Sounds like it will, let's try it!



And that actually works fairly well. It gives us this, where we have a unit and basic missiles from the war cry skill appearing in the correct locations in 3D. But what happens when height gets involved?



Diablo II can convince you that you're on a staircase by blocking off the sides, only allowing you to walk on the "stairs", and painting the stairs to look like, well, stairs. But we're going to have *actual stairs*. They can't be flat, because the lighting won't work right, and we did commit to making a 3-dimensional game. So... what are we to do?



When we first thought about this problem, we considered and prototyped a solution which applied texture mapping to translate between 2D and 3D. The idea was there would be an underlying navigational mesh which was texture mapped with a “fake texture” that was the top-down planar space of the 2-dimensional game.

When we have the location where a unit is in 2D game space, we can put that into local coordinates of the texture space, then convert that to barycentric coordinates on the navigation mesh, and finally use those coordinates to derive the 3D position. We were going to move forward with this idea until a little later on in development when we looked at it a different way.



We considered that we had a fixed camera, that's not user controlled, and a pillar of the project was to have a 1:1 correspondence between 3D and 2D elements on the screen. If that holds true, and we still consider the 2D game to take place on a flat plane in a pseudo-3D space, then we can fire a ray into the screen at the position of, well, anything, and determine where that ray intersects our 3D world, then place that thing at that position in the 3D world.

In this video everything in red is the location of the 2D game units on the 2D game plane. Rays are fired into the scene at each location and collide with the 3D floor, where the green 3D units are placed. This ends up guaranteeing that the 3D elements appear at the same screen location as the 2D sprites.

```

// We need to use the game camera to be able to adjust our translated world position to account for the view direction
// If we don't we end up placing the units at a position where they would be in the 2D game on the flat plane. However
// that method falls apart when we add elevation into the equation. So in this case we are projecting the world position
// back up to the new XY position that the game "sees" as the old location.
const float kRayDistance = 1000.0f;
bcVec4A cameraViewRay = bcMul(kRayDistance, Camera::GameCameraViewVector);
bcVec4A traceFromPoint = bcAdd(worldPosition, cameraViewRay);
bcVec4A castThroughPoint = bcSub(worldPosition, cameraViewRay);

//
//           o trace from point
//          /
//         /
//        /
// -----X----- simPoint on simworld
//        \
//       \
//      \
//           o cast through point
//

```

This is what that looks like in code, with some surprisingly helpful comments and ascii art. We take the position that we are trying to convert, establish it as our trace point, get the camera view ray, and cast to find an intersection point in the 3D world.

```

// We need to use the game camera to be able to adjust our translated world position to account for the view direction
// If we don't we end up placing the units at a position where they would be in the 2D game on the flat plane. However
// that method falls apart when we add elevation into the equation. So in this case we are projecting the world position
// back up to the new XY position that the game "sees" as the old location.
const float kRayDistance = 1000.0f;
bcVec4A cameraViewRay = bcMul(kRayDistance, Camera::GameCameraViewVector);
bcVec4A traceFromPoint = bcAdd(worldPosition, cameraViewRay);
bcVec4A castThroughPoint = bcSub(worldPosition, cameraViewRay);

//
//      o trace from point
//
//
//      -----X----- simPoint on simworld
//
//
//      /
//      o cast through point

```

This is what that looks like in code, with some surprisingly helpful comments and ascii art. We take the position that we are trying to convert, establish it as our trace point, get the camera view ray, and cast to find an intersection point in the 3D world.


```

// We need to use the game camera to be able to adjust our translated world position to account for the view direction
// If we don't we end up placing the units at a position where they would be in the 2D game on the flat plane. However
// that method falls apart when we add elevation into the equation. So in this case we are projecting the world position
// back up to the new XY position that the game "sees" as the old location.
const float kRayDistance = 1000.0f;
bcVec4A cameraViewRay = bcMul(kRayDistance, Camera::GameCameraViewVector);
bcVec4A traceFromPoint = bcAdd(worldPosition, cameraViewRay);
bcVec4A castThroughPoint = bcSub(worldPosition, cameraViewRay);

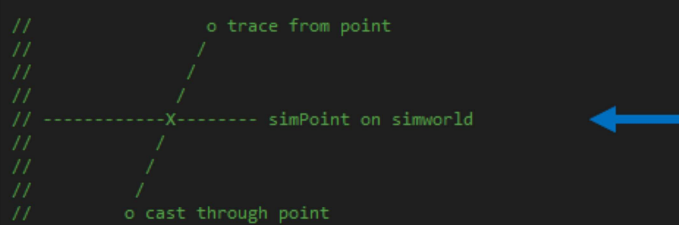
//
//      o trace from point
//
//      /
//     /
//    /
//   /
//  /
// /-----X----- simPoint on simworld
// \
//  \
//   \
//    \
//     \
//      \
//      o cast through point

```

This is what that looks like in code, with some surprisingly helpful comments and ascii art. We take the position that we are trying to convert, establish it as our trace point, get the camera view ray, and cast to find an intersection point in the 3D world.

```
// We need to use the game camera to be able to adjust our translated world position to account for the view direction
// If we don't we end up placing the units at a position where they would be in the 2D game on the flat plane. However
// that method falls apart when we add elevation into the equation. So in this case we are projecting the world position
// back up to the new XY position that the game "sees" as the old location.
const float kRayDistance = 1000.0f;
bcVec4A cameraViewRay = bcMul(kRayDistance, Camera::GameCameraViewVector);
bcVec4A traceFromPoint = bcAdd(worldPosition, cameraViewRay);
bcVec4A castThroughPoint = bcSub(worldPosition, cameraViewRay);

//
//      o trace from point
//
//
//      o cast through point
//
//      -----X----- simPoint on simworld
```

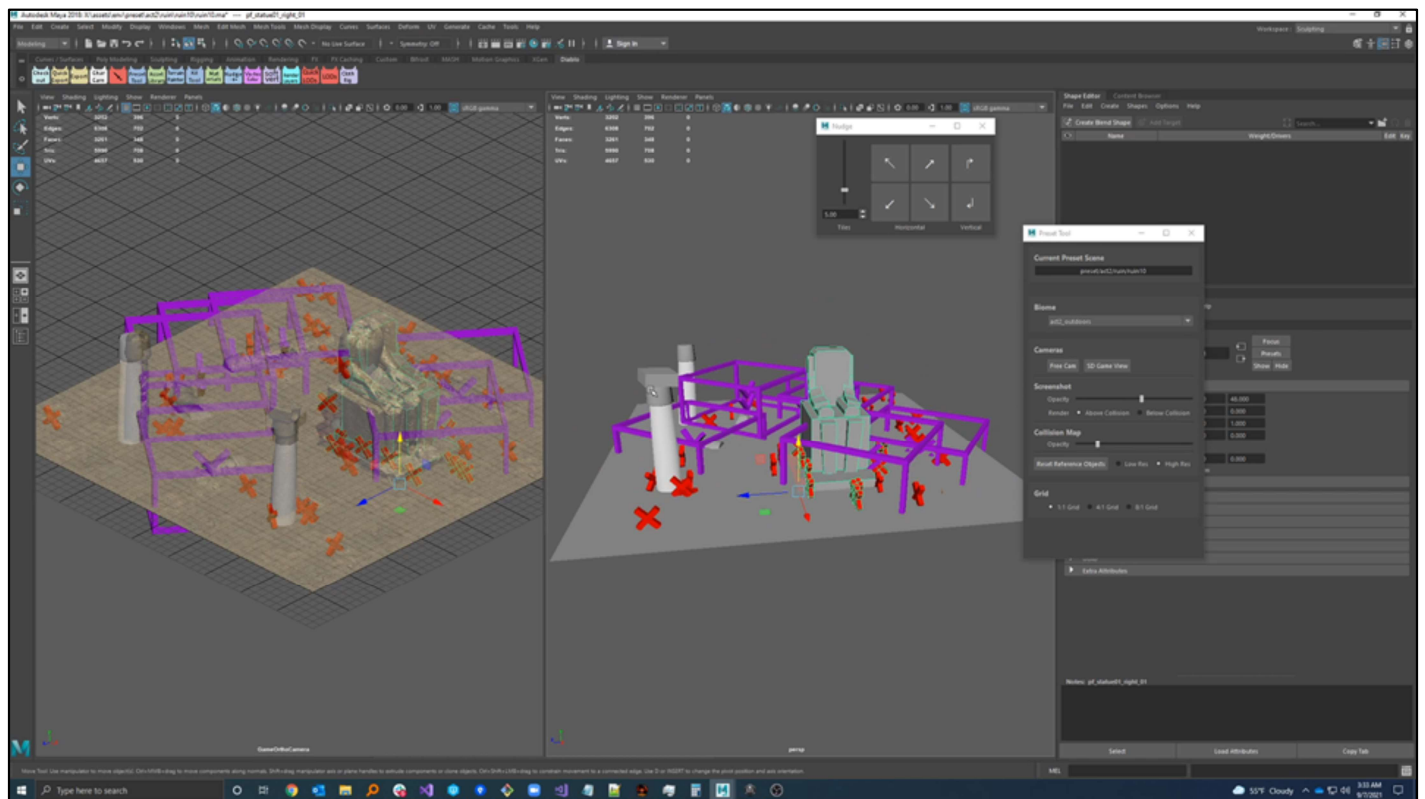


This is what that looks like in code, with some surprisingly helpful comments and ascii art. We take the position that we are trying to convert, establish it as our trace point, get the camera view ray, and cast to find an intersection point in the 3D world.

Orthographic Benefits

- Anything moved along the camera forward vector in an orthographic projection will not appear to move on the projection plane.
- We can accomplish height changes without deviating from a sprite position using this trick.
- Any height change must be accompanied by a shift on the XZ plane.

This technique has some great advantages. First off, the game is generally presented with an orthographic camera, and that means that anything moved along the camera forward vector will not appear to move at all on the projection plane. We'll see a more concrete example of this in just a second. It means that we can accomplish height changes, without deviating 3D objects from their corresponding sprite's position. It just means that any height change has to be accompanied by a shift on the XZ plane.



We actually implemented a custom tool for our artists to do this reliably, which we called the nudge tool. You can see here in Maya an artist is able to nudge an object along the camera view vector, which will alter the height that it sits at, but on the left, its location on screen does not change.



This does end up making some dangerously steep staircases. Arguably there's more danger in this staircase than in the Chaos Sanctuary on Hell difficulty. Anyway...

Checkpoint

- We need a 3D engine.
 - We have a 3D space defined.
 - We can position things in the 3D world.
- We need a toolchain.
- We need the right rendering technology.
- How do we populate it?

Checkpoint. We're still building our 3D engine. We've defined our 3D space and we can position things in the 3D world. How are we going to populate it?



All we really have at this point is a completely empty void. We started to fill this in with debug lines and boxes. We even gave the boxes some character by making them bob as they moved around and tilt as they “attacked.” This was the birth of the layer of the engine which we simply called the “translation layer.”



If we look at the software structure of Diablo II, as you might expect it had different libraries each responsible for different things. For example, there was Client, which handled most of the client-side functionality and communication with the server; Common, which contained a pool of common functionality for all the engine; Game, which as you might guess contained the bulk of the logic for the game itself; and Gfx which of course handled the drawing of the sprites.

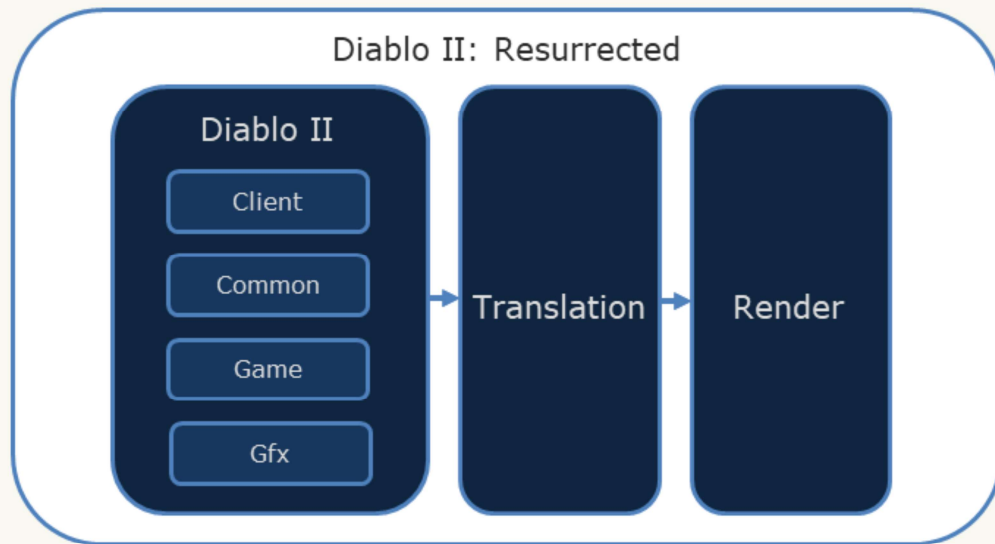
Units

- Everything is a unit.

2	name	compactsave	version	level	ShowLevel	levelreq	reqstr	reqdex	rarity	spawnable										
3	Elixir	1	0	21	0	0	4	1	0	1	20	elx	elx	elx	16	1	0	0	0	0
4	Healing Potion	1	0	0	0	0	1	0	0	1	30	hpo	hpo	hpo	16	1	1	0	0	0
5	Mana Potion	1	0	0	0	0	1	0	0	1	30	mpo	mpo	mpo	16	1	1	0	0	0
6	Full Healing Potion	1	0	0	0	0	2	0	0	1	150	hpf	hpo	hpf	16	1	1	1	1	1
7	Full Mana Potion	1	0	0	0	0	2	0	0	1	150	mpf	mpo	mpf	16	1	1	1	1	1
8	Stamina Potion	1	0	0	0	0	1	1	0	1	25	vps	vps	vps	16	1	1	0	0	0
9	Antidote Potion	1	0	0	0	0	1	1	0	1	40	yps	yps	yps	16	1	1	0	0	0
10	Rejuvenation Potion	1	0	0	0	0	2	0	0	1	400	rvs	yps	rvs	16	1	1	1	1	1
11	Full Rejuvenation Potion	1	0	0	0	0	2	0	0	1	1500	rvl	ypf	rvl	16	1	1	1	1	1
12	Thawing Potion	1	0	0	0	0	2	1	0	1	25	wms	yps	wms	16	1	1	0	0	0
13	Tome of Town Portal	0	0	0	0	0	2	1	0	1	250	tbk	hbk	tbk	16	1	2	1	2	2
14	Tome of Identify	0	0	0	0	0	2	1	0	1	200	ibk	rbk	ibk	16	1	2	1	2	2
15	Amulet	0	0	1	0	0	4	1	0	1	2400	63000	amu	amu	16	1	1	1	1	1
16	Amulet of the Viper	0	0	15	0	0	4	0	0	1	400	vip	vip	vip	16	1	1	1	1	1
17	Ring	0	0	1	0	0	4	1	0	1	1800	50000	rin	rin	16	1	1	1	1	1
18	Gold	1	0	0	0	0	1	1	0	1	0	gld	gld	gld	16	1	1	0	0	0
19	Scroll of Infuse	1	0	0	0	0	0	0	0	1	100	bks	bks	bks	16	2	2	2	2	2
20	Key to the Cairn Stones	1	0	0	0	0	0	0	0	1	12000	bkd	bkd	bkd	16	1	1	1	1	1
21	Arrows	0	0	0	0	0	1	1	0	1	256	aqv	aqv	aqv	16	1	3	0	0	0
22	Torch	1	0	0	0	0	0	0	0	1	50	tch	bsh	tch	7	1	2	0	0	0
23	Bolts	0	0	0	0	0	1	1	0	1	256	cqv	cqv	cqv	16	1	3	0	0	0
24	Scroll of Town Portal	1	0	0	0	0	2	1	0	1	100	tsc	bsc	tsc	16	1	1	1	1	1
25	Scroll of Identify	1	0	0	0	0	2	1	0	1	80	isc	rsc	isc	16	1	1	1	1	1

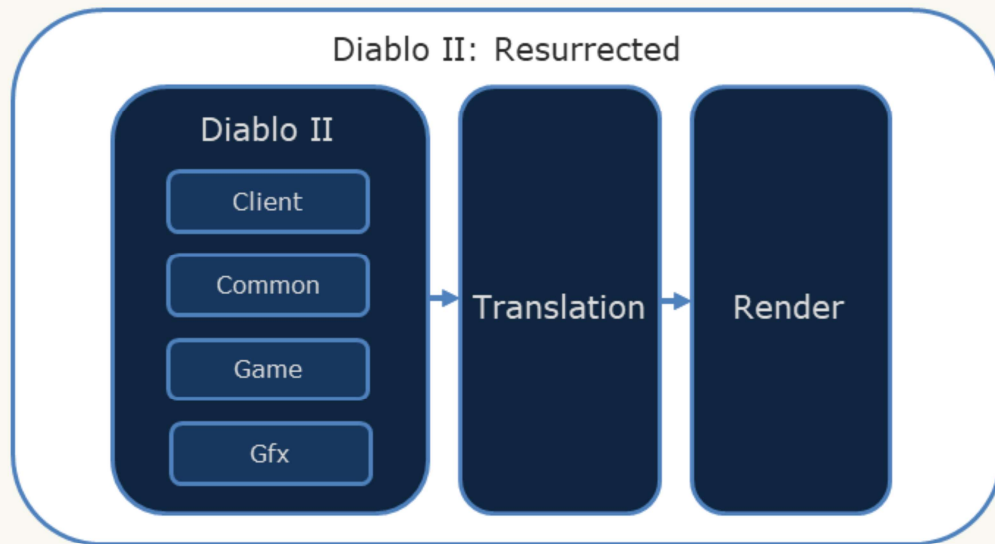


Now, every functional piece of the game – items, characters, objects like chests and shrines, missiles like arrows and spells – are all called Units. A unit has a type that can be any one of those things I just mentioned. All of the game's content is controlled by CSV files that dictate what each unit is, what its stats are, where it spawns and in what number, etc. Abilities are hardcoded, but can spawn other units.



For Resurrected, our idea was straightforward – create a mapping of every unit in the game to a 3-dimensional counterpart. Any time that a particular unit is created, find the corresponding 3D assets and load them, then start updating them according to the unit's state. This is where the translation layer came from. This software layer sits in between the original game and the new library built from scratch to manage 3D rendering. In practice it's quite a bit of code, but in theory it's a shim between the logical components of the original game and the visuals of the remaster. This is what does all the heavy lifting of the conversion between 2 dimensional space and 3 dimensional space. All that math we covered in the beginning? That happens here.

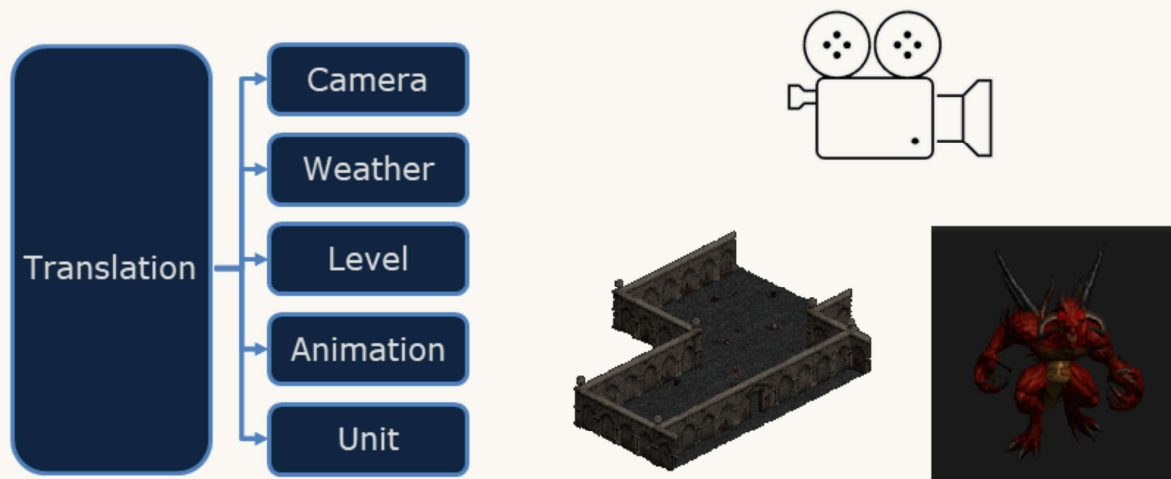
The collection of all these things make up the software package which is Resurrected.



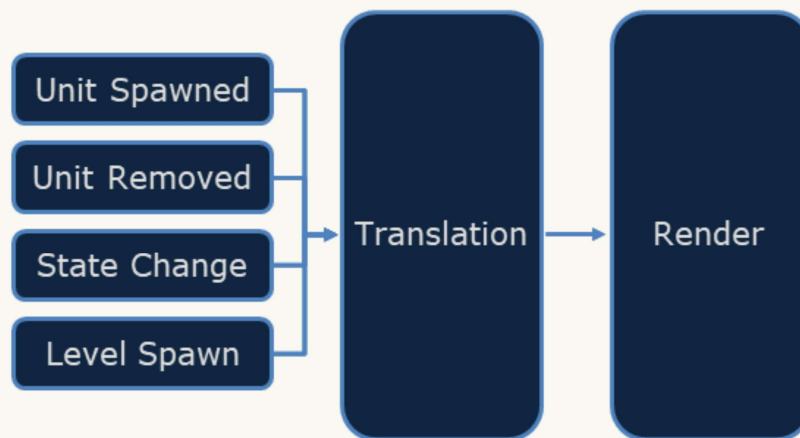
For Resurrected, our idea was straightforward – create a mapping of every unit in the game to a 3-dimensional counterpart. Any time that a particular unit is created, find the corresponding 3D assets and load them, then start updating them according to the unit's state. This is where the translation layer came from. This software layer sits in between the original game and the new library built from scratch to manage 3D rendering. In practice it's quite a bit of code, but in theory it's a shim between the logical components of the original game and the visuals of the remaster. This is what does all the heavy lifting of the conversion between 2 dimensional space and 3 dimensional space. All that math we covered in the beginning? That happens here.

The collection of all these things make up the software package which is Resurrected.

Translation



Now in reality, the layer had several specializations besides just units. Things like the camera as I described earlier, for effects like camera shake and horizontal offsets when menus are opened; weather state, which operates outside of the “unit” structure; level translation for the placement of rooms and the building blocks of physical space in the game; and animation layers all exist in addition to translation of game units.



Generally, the flow would be that any event that would occur in the original game would push a message to the translation layer that some action needed to take place, such as spawning or despawning a unit, changing a unit's animation state, or instantiating a level. The translation layer would then perform its management of the runtime objects which had associated render objects for the 3D scene. This was a push method, but the translation layer would also operate on a pull method for other information like "what color is this unit" or "what properties does the light associated with this unit have?"

JSON

- Elected to use JSON for several reasons
 - Mod-ability
 - Off-the-shelf, rapid development
 - Convenience
- Resurrect an old tool from Vicarious – Laboratory
 - Easy compatibility with JSON
 - Relevant expertise
 - Build system
 - Fastest way of going from nothing to having tools

We needed some way to represent all these mappings and the structure of the data for the 3D engine itself. For this we elected to use the JSON format, and there were several reasons for this. The foremost was mod-ability. Diablo II has a very passionate modding community, so one of things that we wanted to make sure to carry forward as best as possible was the openness of the content we were developing for the 3D visuals. Thus, we felt that much of our 3D content would need to be in a human readable format in order to make it easy to modify.

The other advantage is in development workflows. It's a lot easier to dig into data that's human readable and solve problems than it is to inspect some broken binary data. On top of that, it enables much of the art and design teams to be more hands on with troubleshooting data files.

And lastly, it was just highly strategic move for us. It allowed us to adopt an existing toolset from our old engine at Vicarious Visions – Alchemy Laboratory. Compatibility with JSON was easy to add, we had relevant expertise on the team, and it offered us a full build harness to start writing scripts for our databuild. It was the fastest way of going from essentially nothing, as far as infrastructure was concerned, to having a full toolset.

JSON

- 7500+ JSON files
 - Animation state
 - Presets
 - Prefabs
 - Effects
 - Objects
 - Characters
 - Visual Data
 - Biomes

```
1 {  
2   "dependencies": {  
3     "particles": {  
4       },  
5     "models": {  
6       {  
7         "path": "data/bd/env/model/global/prop/act3/jungle/act3_jungle_rocks/rock_pile02.model"  
8       },  
9       {  
10        "path": "data/bd/env/model/global/prop/act3/jungle/act3_jungle_rocks/rock03.model"  
11      },  
12      {  
13        "path": "data/bd/env/model/global/prop/act3/jungle/act3_jungle_rocks/rock04.model"  
14      },  
15      {  
16        "path": "data/bd/env/model/global/prop/act3/jungle/act3_jungle_rocks/rock05.model"  
17      },  
18      {  
19        "path": "data/bd/env/model/global/prop/act3/jungle/act3_jungle_rocks/rock06.model"  
20      }  
21    },  
22    "skeletons": {  
23      },  
24    },  
25    "animations": {  
26      },  
27    },  
28    "textures": {  
29      {  
30        "path": "data/bd/env/texture/global/prop/act3/act3_rocks_rocks01_alb.texture"  
31      },  
32      {  
33        "path": "data/bd/env/texture/global/prop/act3/act3_rocks_rocks01_nrm.texture"  
34      },  
35      {  
36        "path": "data/bd/env/texture/global/prop/act3/act3_rocks_rocks01_orm.texture"  
37      },  
38      {  
39        "path": "data/bd/env/texture/global/prop/act3/act3_rocks_rocks02_alb.texture"  
40      },  
41      {  
42        "path": "data/bd/env/texture/global/prop/act3/act3_rocks_rocks02_nrm.texture"  
43      },  
44      {  
45        "path": "data/bd/env/texture/global/prop/act3/act3_rocks_rocks02_orm.texture"  
46      }  
47    },  
48    "physics": {  
49      },  
50    },  
51  }
```

The game has over 7500 json files to represent 3D assets. Everything from animation state machines, presets, prefabs, effects, objects, characters, visual data, and biomes are represented by json.

JSON at Runtime

- 40MB dedicated to loaded JSON
- Plain text is never ideal for load speeds
 - Though not as impactful as instantiation of objects
- Optimized by removing whitespace
- Parsing can cause heavy fragmentation
 - Sequestered into its own memory pool

At runtime we have 40 MB of memory dedicated to loaded json files. Now, as you might expect plain text is never the friendliest to load speeds. In practice, we had the most time spent in instantiation of objects, in part because of the larger scope of logic required to set up an object based on properties gathered from the original simulation.

We optimized for load speeds by removing whitespace in built json files.

Another issue we had was severe fragmentation caused by the parsing of the plain text, so we sequestered json loading into its own pool of memory that could effectively be cleared between loads.

JSON, if we tried again

- Limit information to what's changed
 - We didn't strip out default values
- Better dependency checking
 - Not robust
 - Pulls in more than it has to
- Binary compression
 - At odds with human-readable modability
 - Much faster to read and instantiate
- Probably not use JSON.

If we were to do this all over again, there are further steps we could take. A big one is limiting the amount of information actually stored in the file. We didn't end up stripping out default values from our files because at the time it became relevant, it was too risky to do on all the game's data.

We could also explore better dependency checking. Our records of dependencies are not robust and we likely load more data in dependencies than we really need to during parts of the game. In many cases we fell back on hand optimization of dependency loading in order to resolve this situation. An advantage we had because we knew what the full scope of the game was and how it would be played from the start.

Switching to something more obvious, we would move to a binary compressed version for processed game files instead. This will give us that edge in load times and instantiation times.

But honestly, we probably would just not use json if we were to do this again.

Where are we?

- We need a 3D engine.
- We need a toolchain.
- We need the right rendering technology.

Right, so – we have our 3D engine now! We can associate and load 3D assets with original game content, position them, move them, animate them – everything needed to make gameplay happen.

Creating Content

- We have a crude 3D space
- Art needs to start working
- Diablo II is highly procedural
 - So, what should we have them make?

How are we going to make all this content? What are we going to have our art teams do? And in particular, what about environments? Units in the game like missiles and characters and objects are one thing, but procedural level building is a major component of the game.



One solution we considered was very similar to how things were originally done: what if we built a bunch of 3D content and then created an automated process to slice it up into tiny pieces that correspond to all the sprites?



Then we could draw all the pieces in the same manner that the sprite game draws to reassemble the original 3D model.

...Yea, that's definitely too crazy. Some of the cons to this were potentially long processing times -- something we wanted to avoid because we needed to develop very rapidly -- and uncertainty about what we would do with LoDs; texture seams; lighting seams; high draw call count; you name it, it seemed a little too crazy overall.



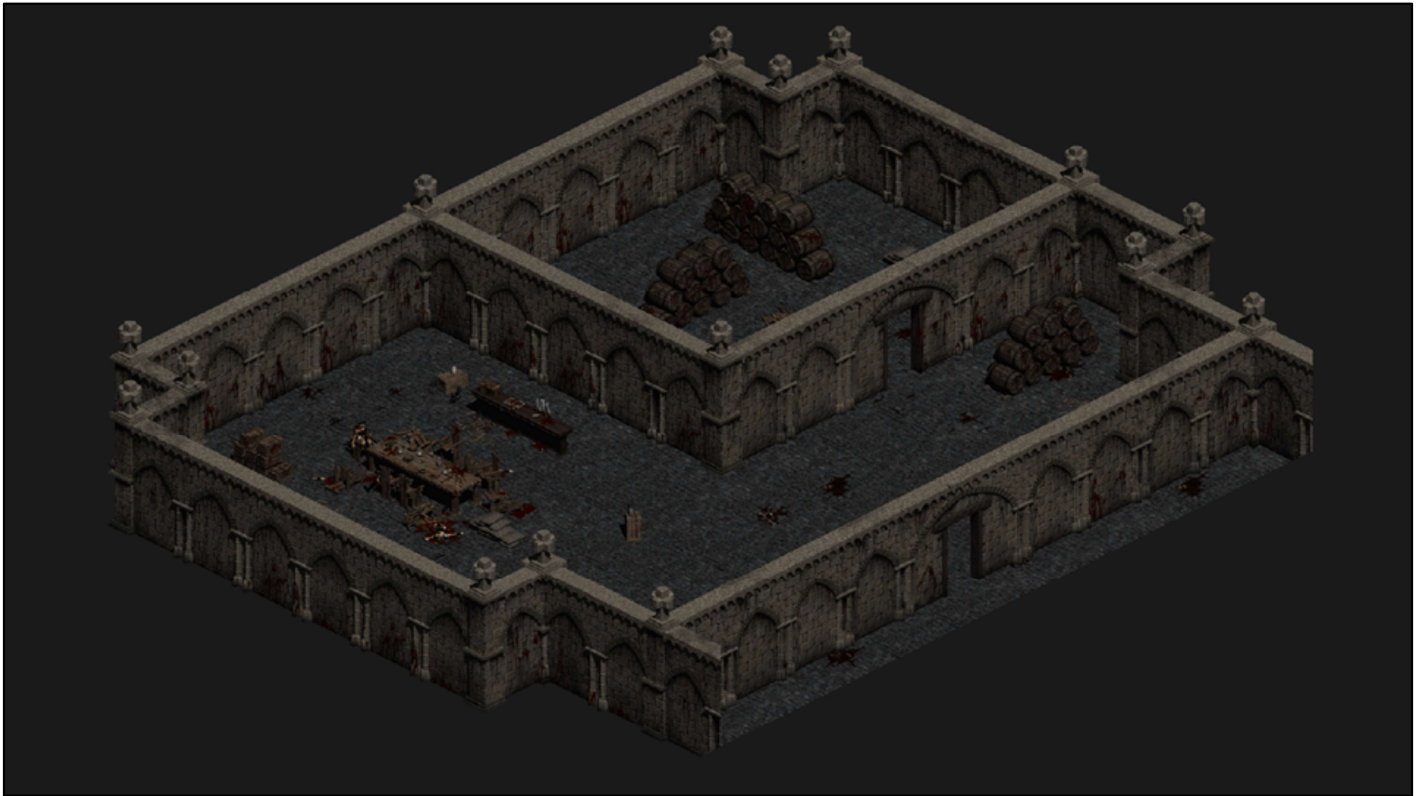
[image courtesy GameSpot]

Too crazy even for the studio that brought you a 3D game on a Game Boy Advance.

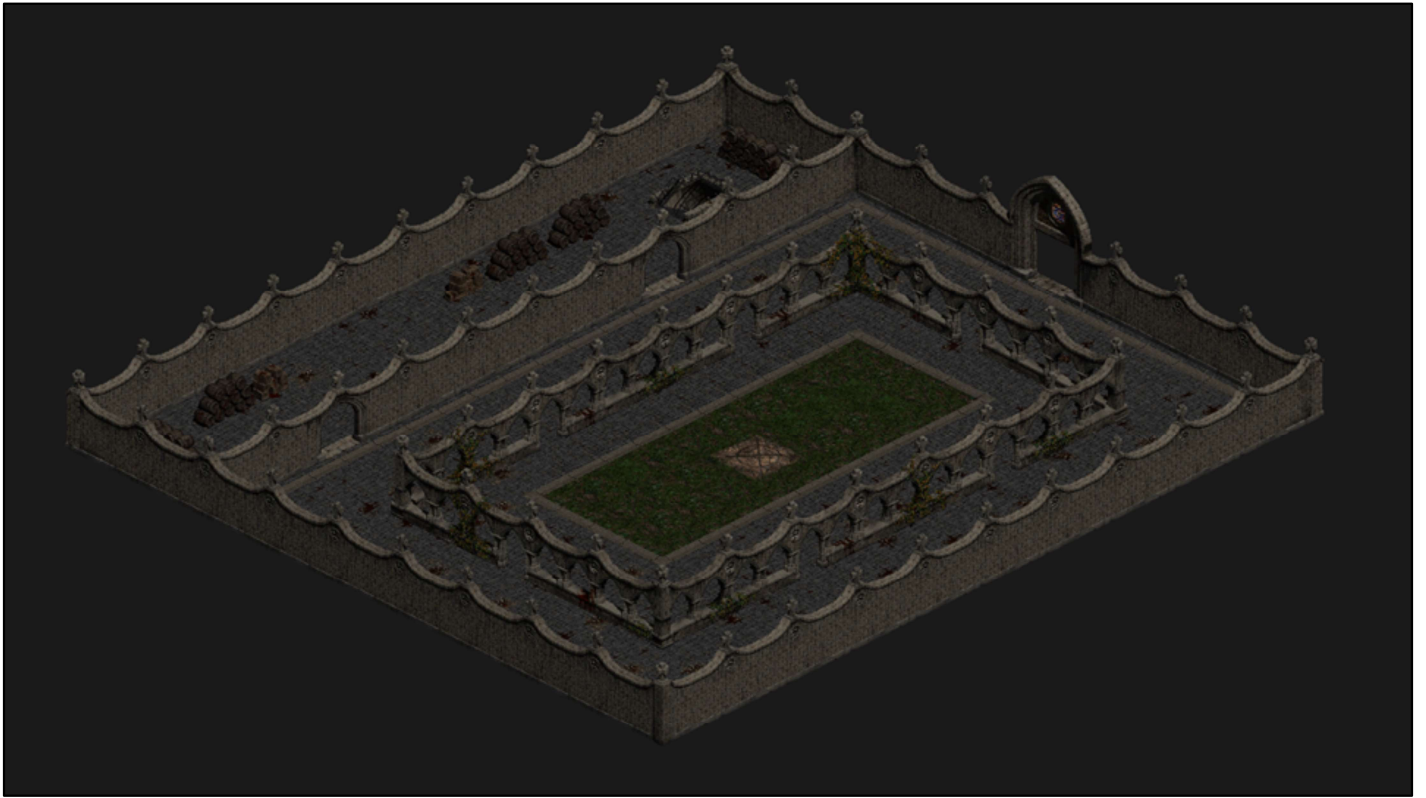
Presets

- Building block of procedural spaces
 - Made up themselves of tiles
- Presets can vary in size
- Outdoor spaces have some special rules

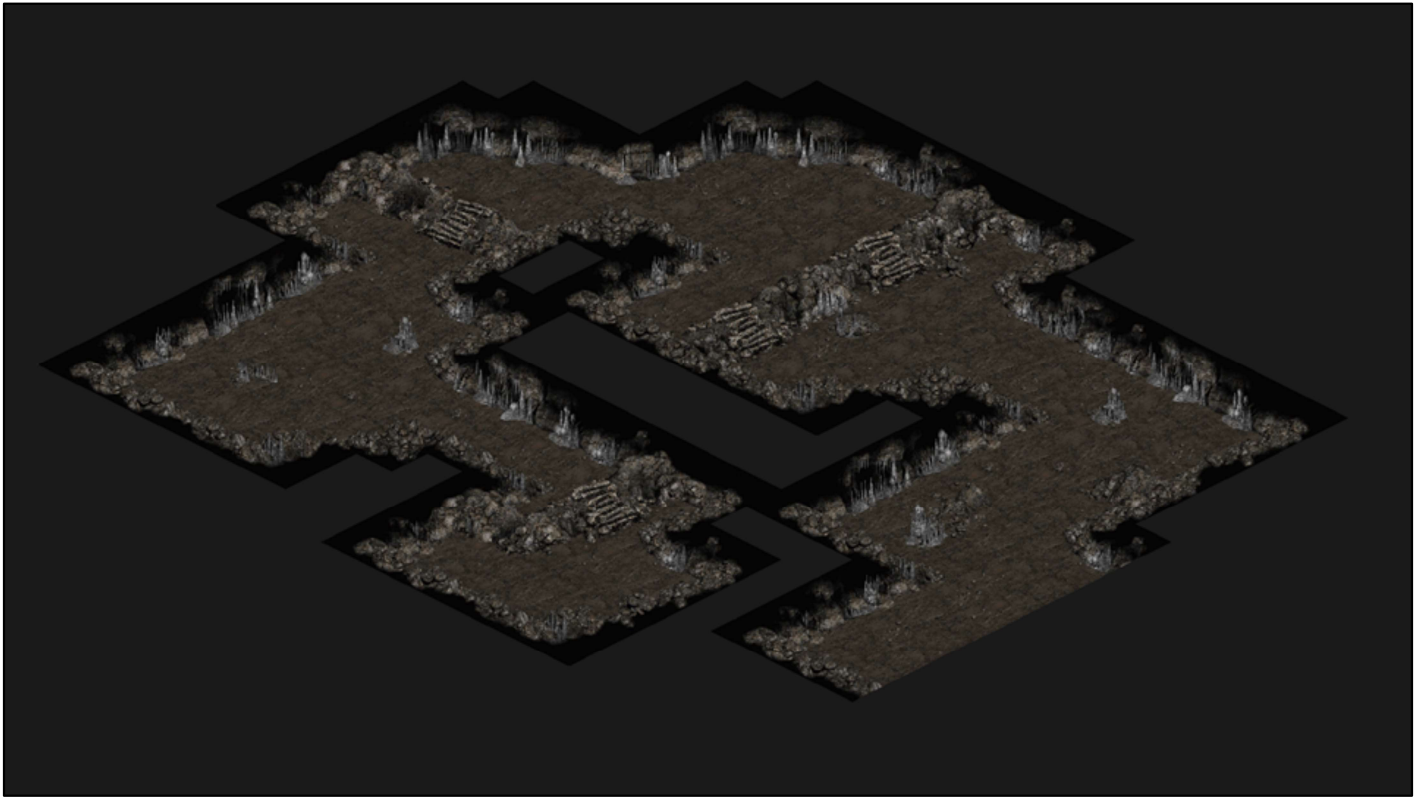
To solve this problem, we took a deeper look at how the Diablo II engine works. Diablo II assembles levels from a collection of building blocks called presets. Presets are assembled from a collection of tiles, but we weren't looking to go to that level of granularity. Presets come in all shapes and sizes, and there are also some special blocks for outdoor areas called clusters that we won't really get into here.



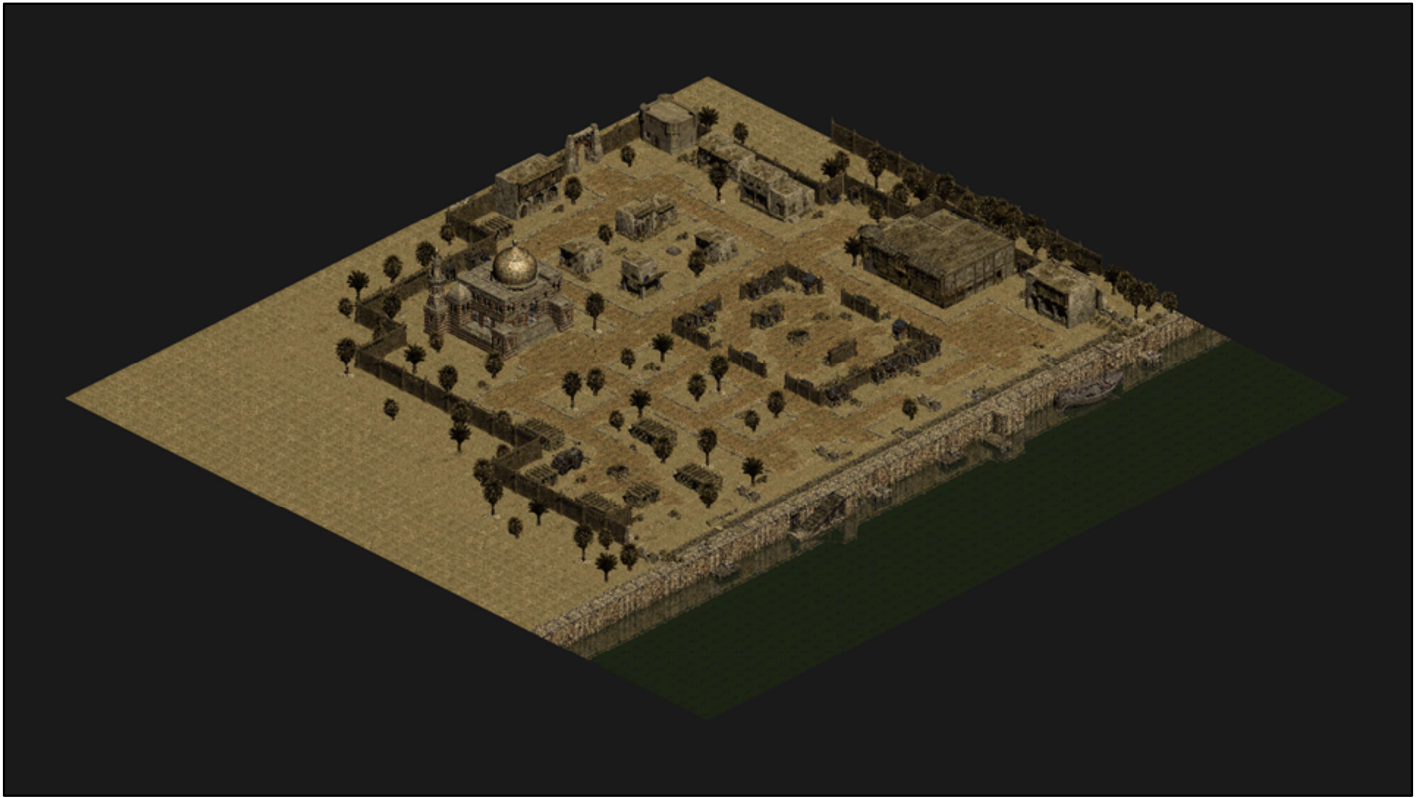
Presets exist for every area of the game, like the barracks



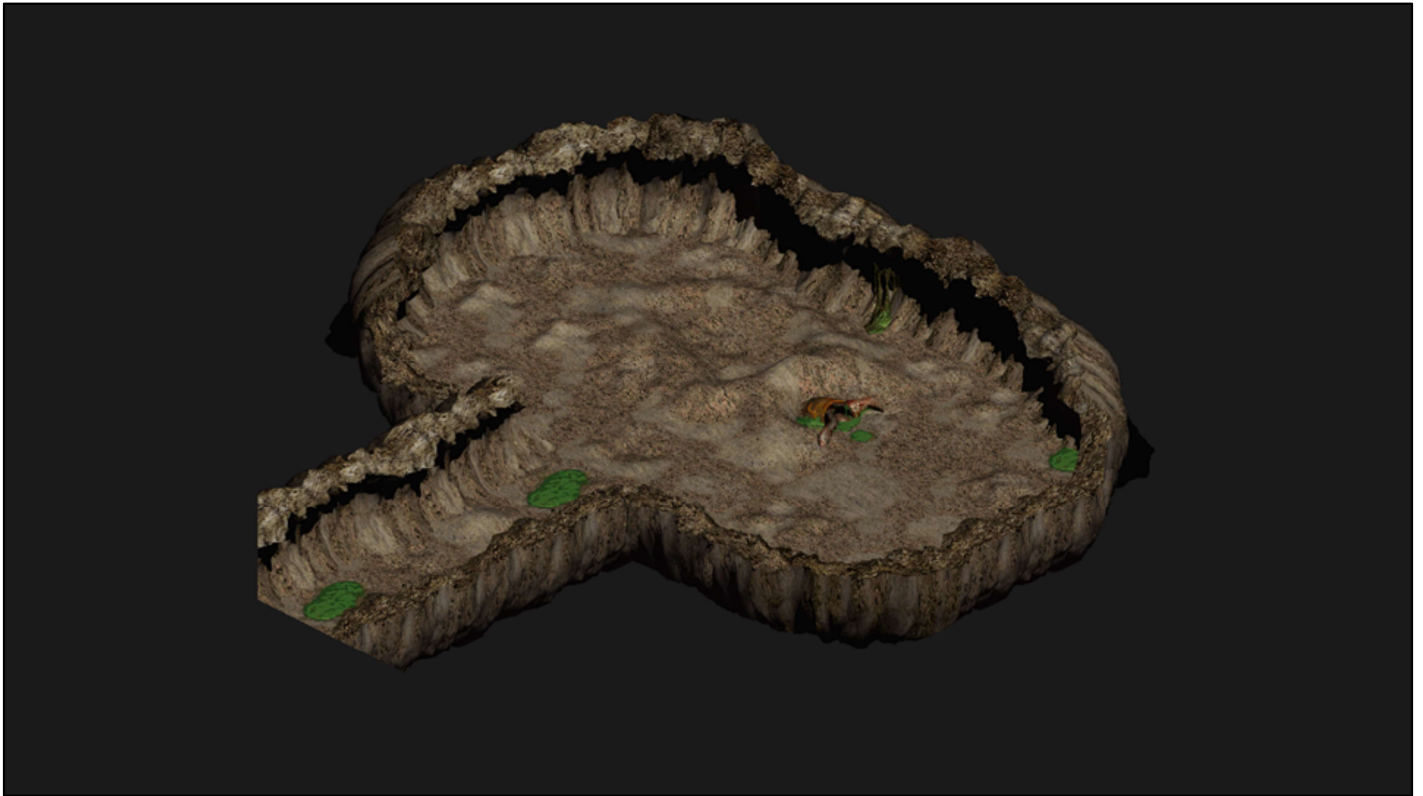
And the inner cloister



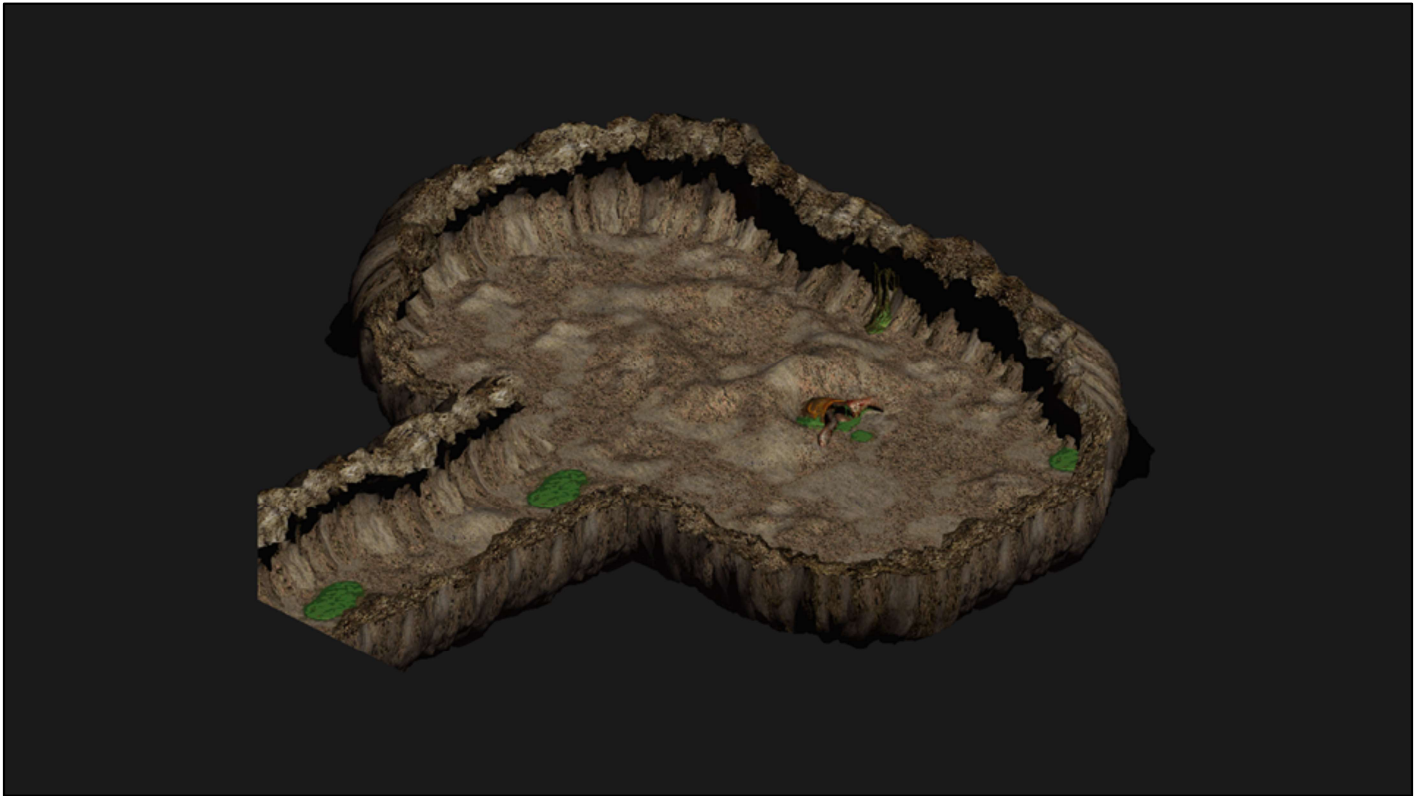
And the caves



And entire towns like Lut Gholein

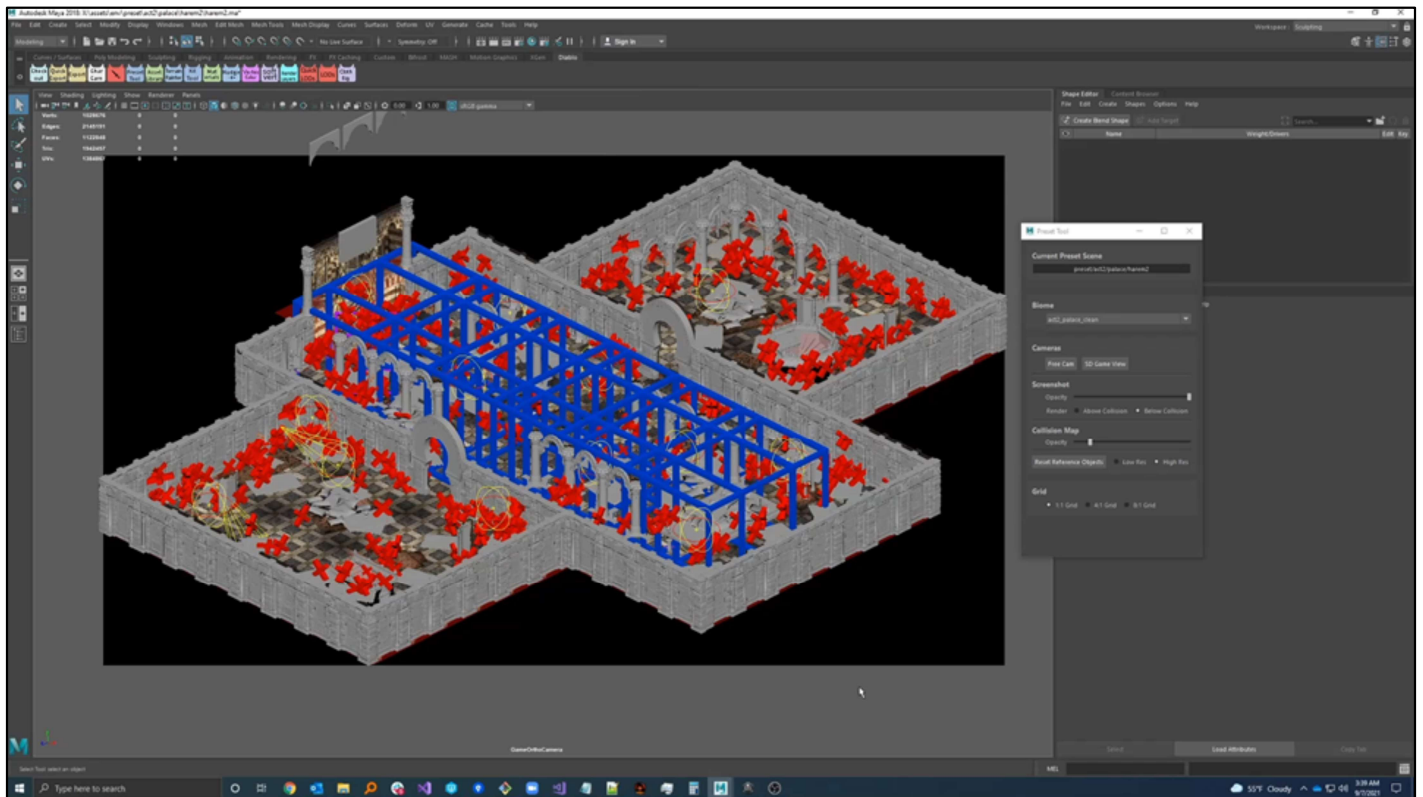


And also fixed locations like the maggot queen's lair. But how many presets are there?



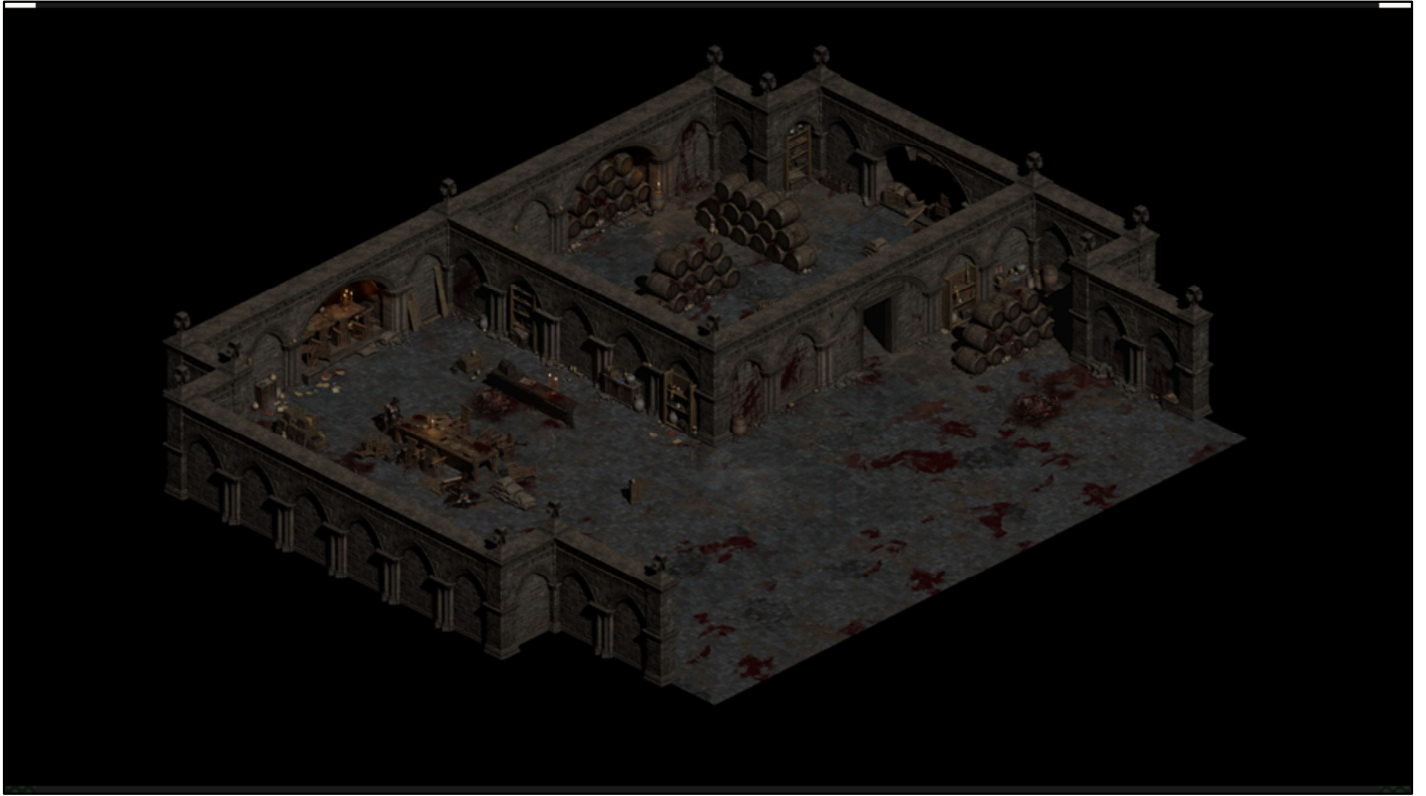
There are exactly two thousand, four hundred, and sixteen presets (2416) in the entire game. This isn't even all of them here on this slide. That's a lot... but maybe not too much when you think about how many assets go into a game these days.

What we can do is have the environment art team create 3D versions of all 2416 presets, and these presets can be placed in the 3D world using the 2D game coordinates where their counterpart presets are spawned.

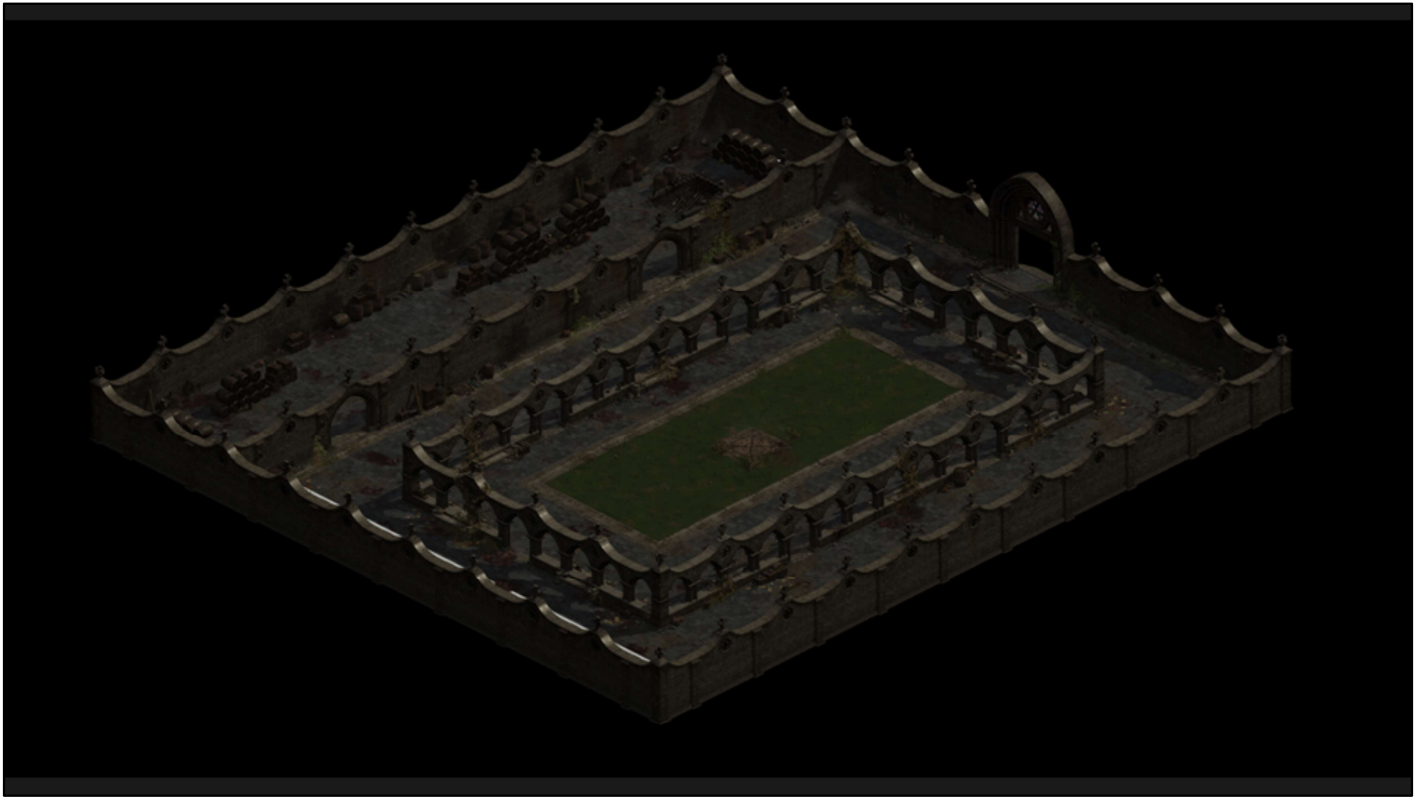


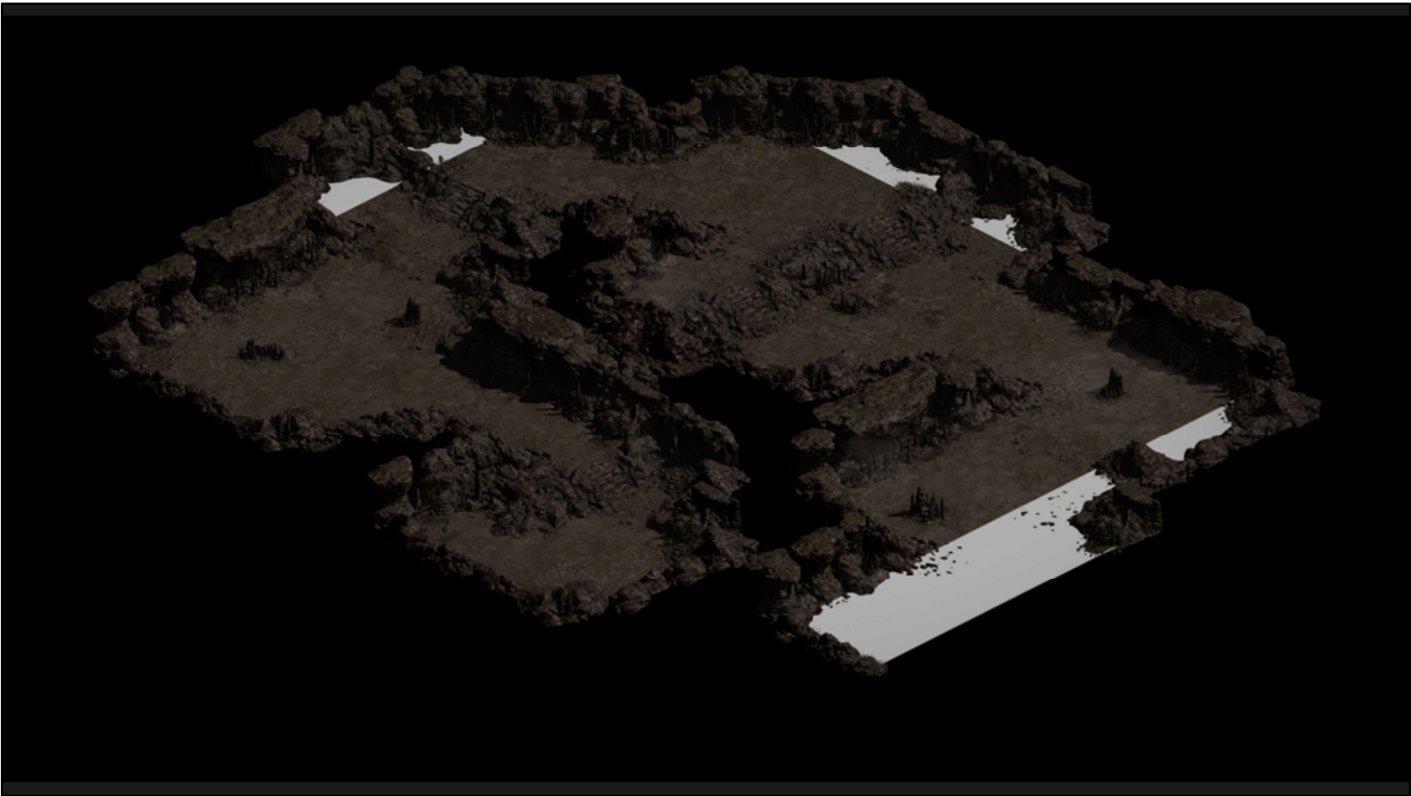
To do this, we created a custom toolset in Maya that fixes the camera with an orthographic projection identical to the game and creates an image plane that shows the 2-dimensional preset over top the 3D geometry as a reference.

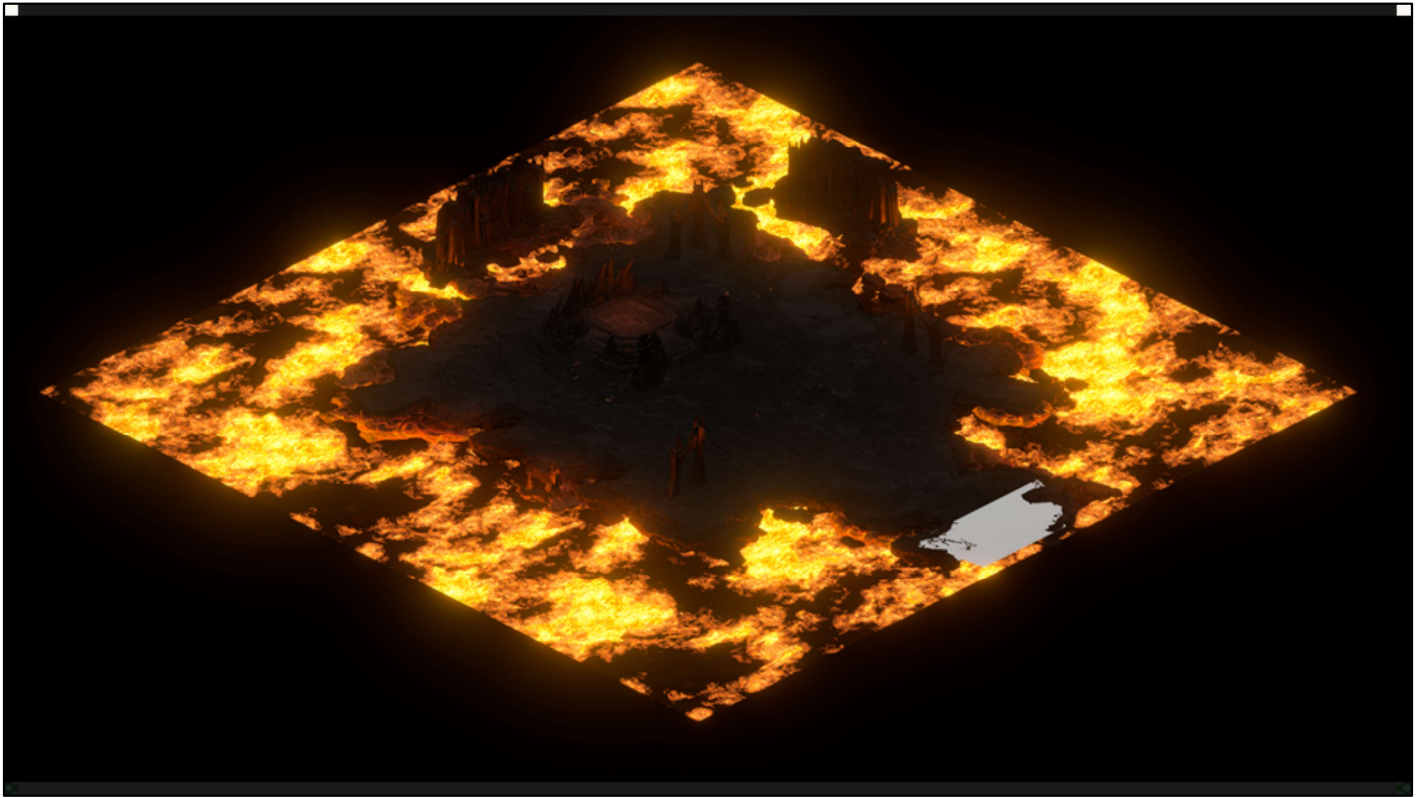
Art would construct 3D presets using separately modeled pieces like pillars, wall segments, torches, and other building blocks all while aligning them with the original.

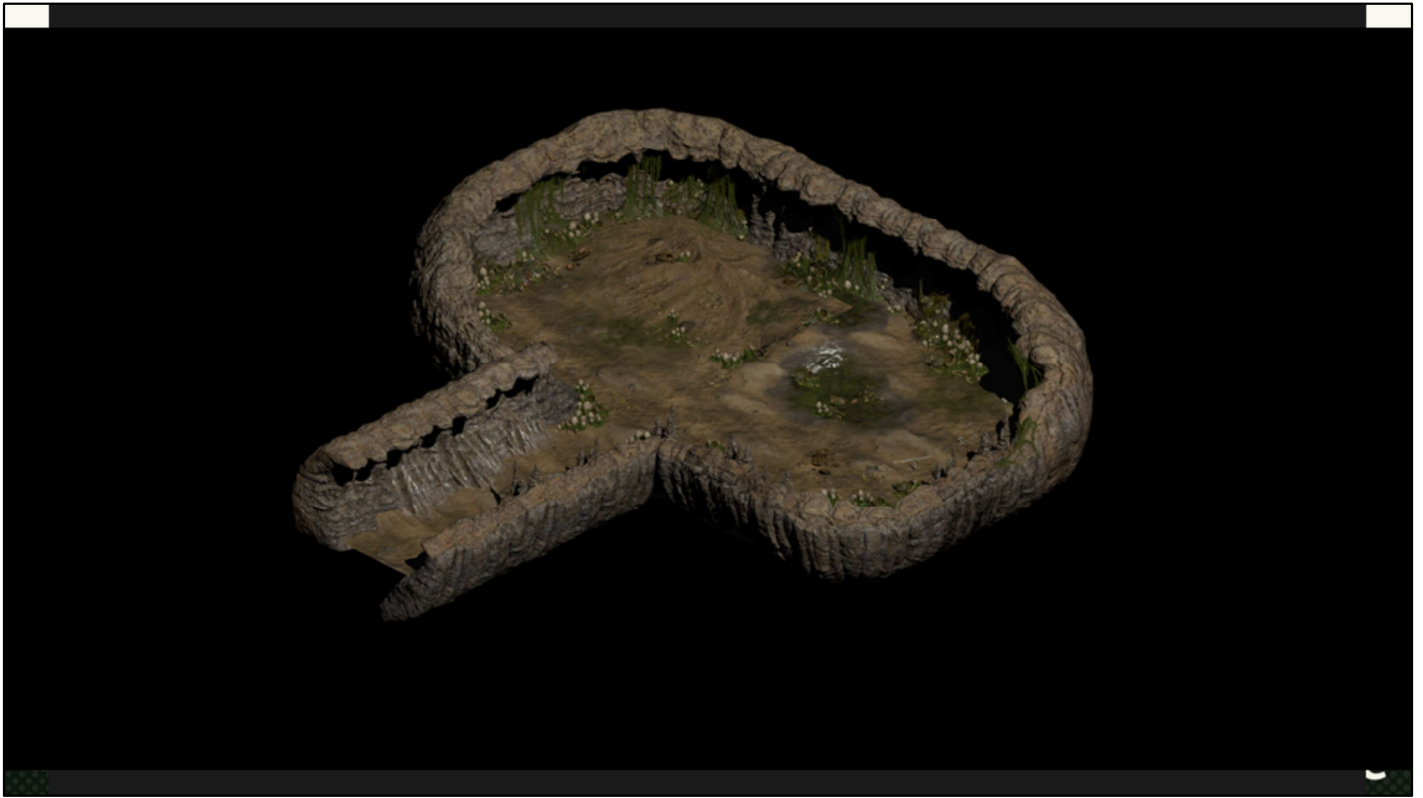


And when that process is done, we can see the transformation of each original preset to a 3D-ified version.









Placing a preset

- Preset coordinates are back left tile
- Multiply tile coordinates by tile size (10 feet)
 - This gives us a location on the XZ plane.
- Height offset
 - Black magic*
 - *we'll cover this shortly
- Adjust XZ position by height

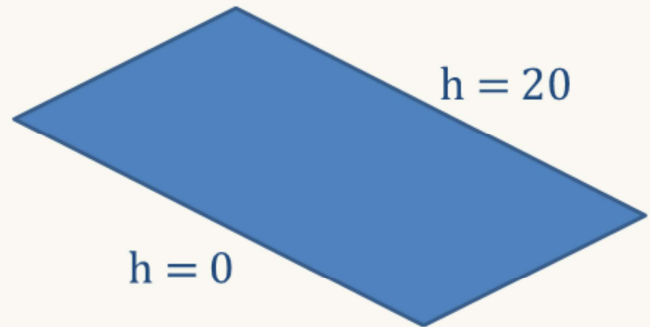
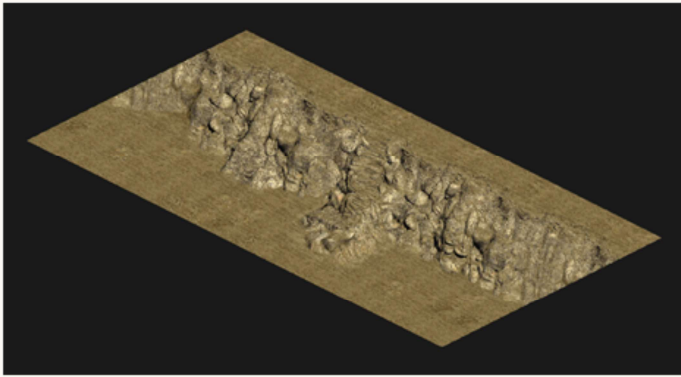
But, we still need to make sure that we can place these presets accurately in the world. And, as we're about to see, it's not quite as simple as placing units.

As we're navigating the world, the game populates the space around us by placing presets that connect together to form the randomly generated dungeons. Via the translation layer, each time a preset is placed, we need to load and create the 3D equivalent.

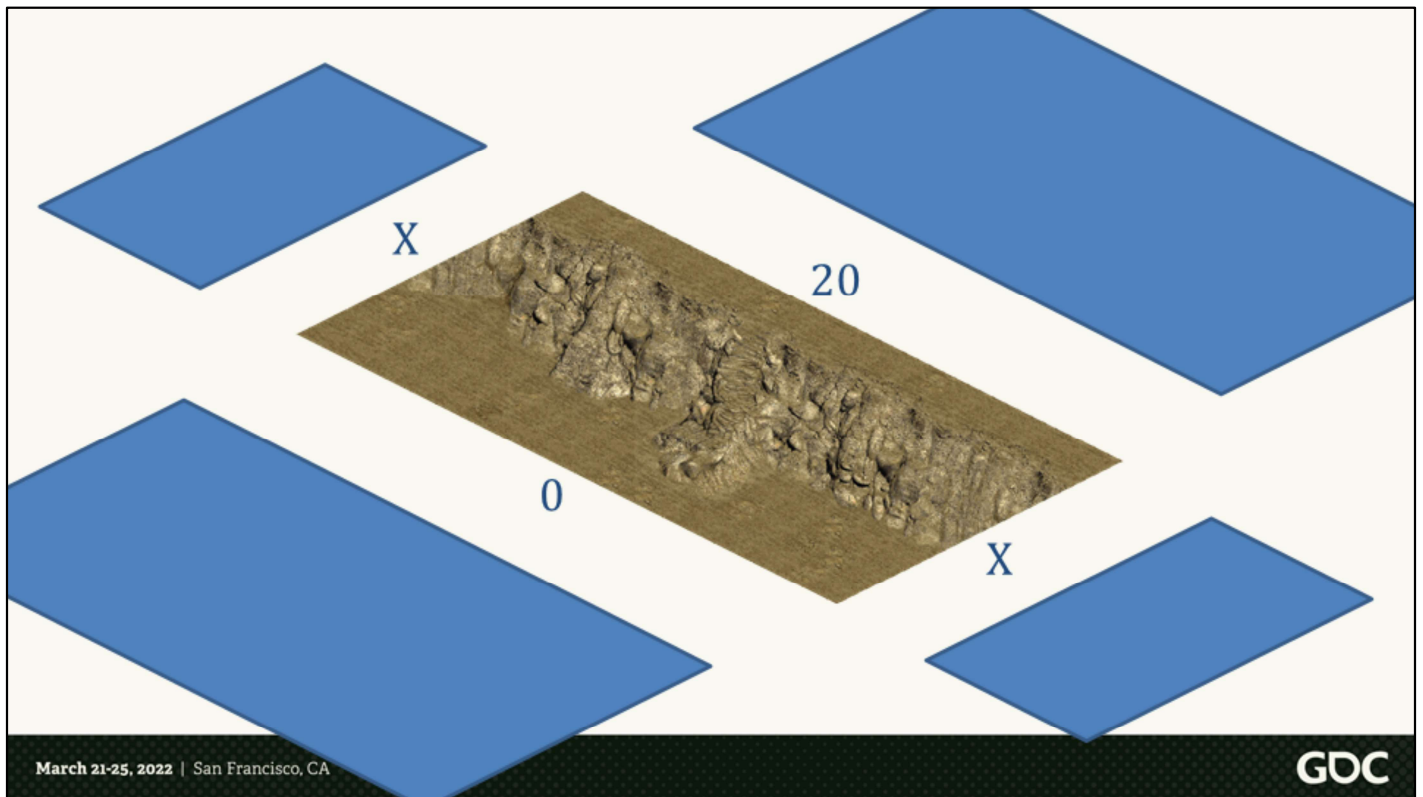
Preset coordinates are given by tile coordinates since every preset is going to be some specific number of floor tiles in size. These tile coordinates can be used directly to calculate our position in world space along the XZ plane. This works great if our world is flat, but we're adding an extra dimension to the game, so we need to consider what height the preset should be at. *That* is the real black magic here and I'll talk about that in just a second. Finally, we'd use this height offset to adjust the position along the XZ plane, since we need to preserve the preset's location on the projection plane.

Preset Height

- Some presets have changes in height
- We need to find a way to align them



Some areas of the game have faked elevation changes, and we're talking about making these real elevation changes. Take this cliff for example. On the left side of the cliff you're standing at height 0, while on the right side after ascending the stairs you're at height 20. Any preset that falls to the right of this one needs to be placed at the correct height, otherwise we'll end up with a discontinuity in the terrain and the environment lighting.

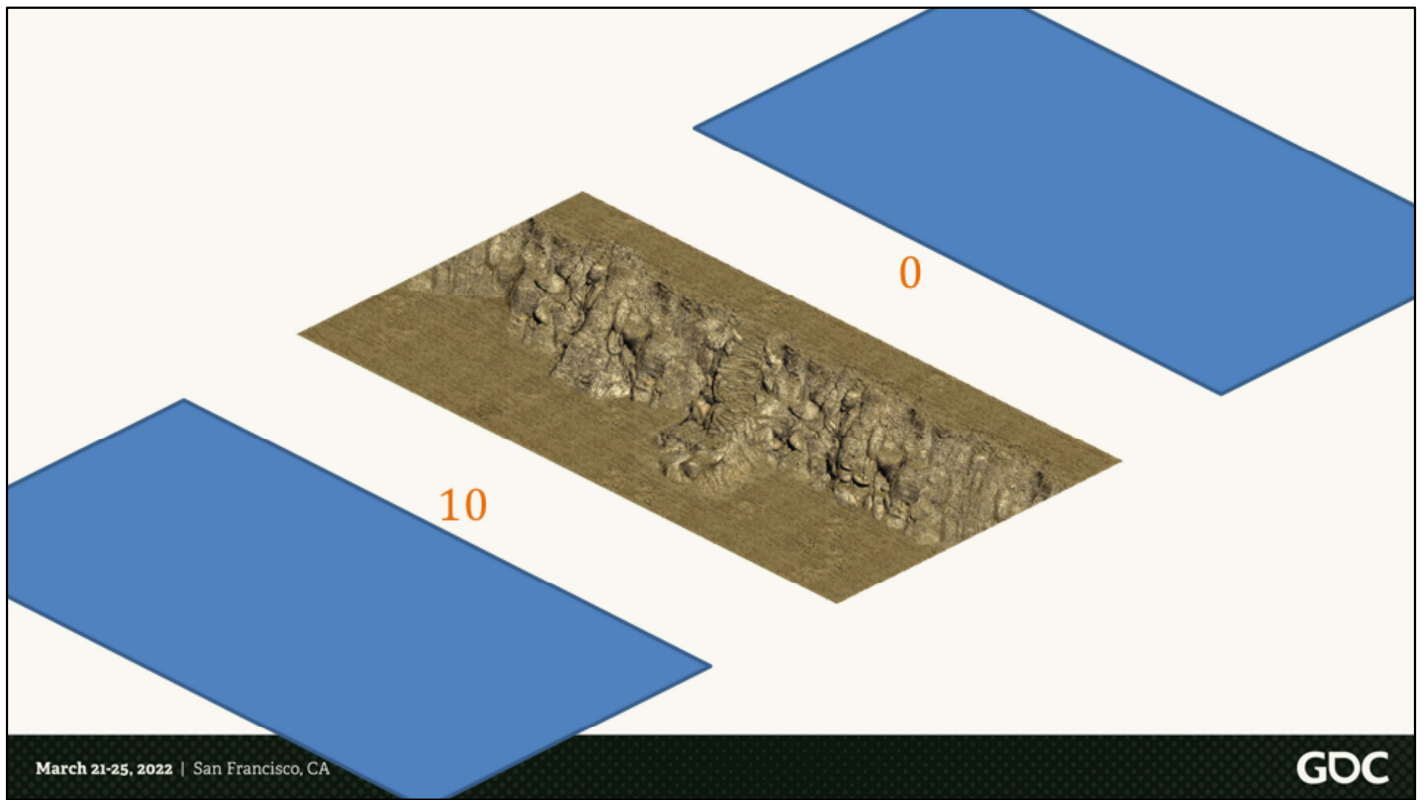


So the algorithm goes something like this: start with a preset that we need to place. Let's say we need to place this desert cliff preset. We have markup for the height of the northeast and southwest edges, the other two, in this case, are irrelevant, because, very kindly of the game, it will always line up sides of presets with height discontinuities, since it wouldn't make much visual sense to break that in 2D either.

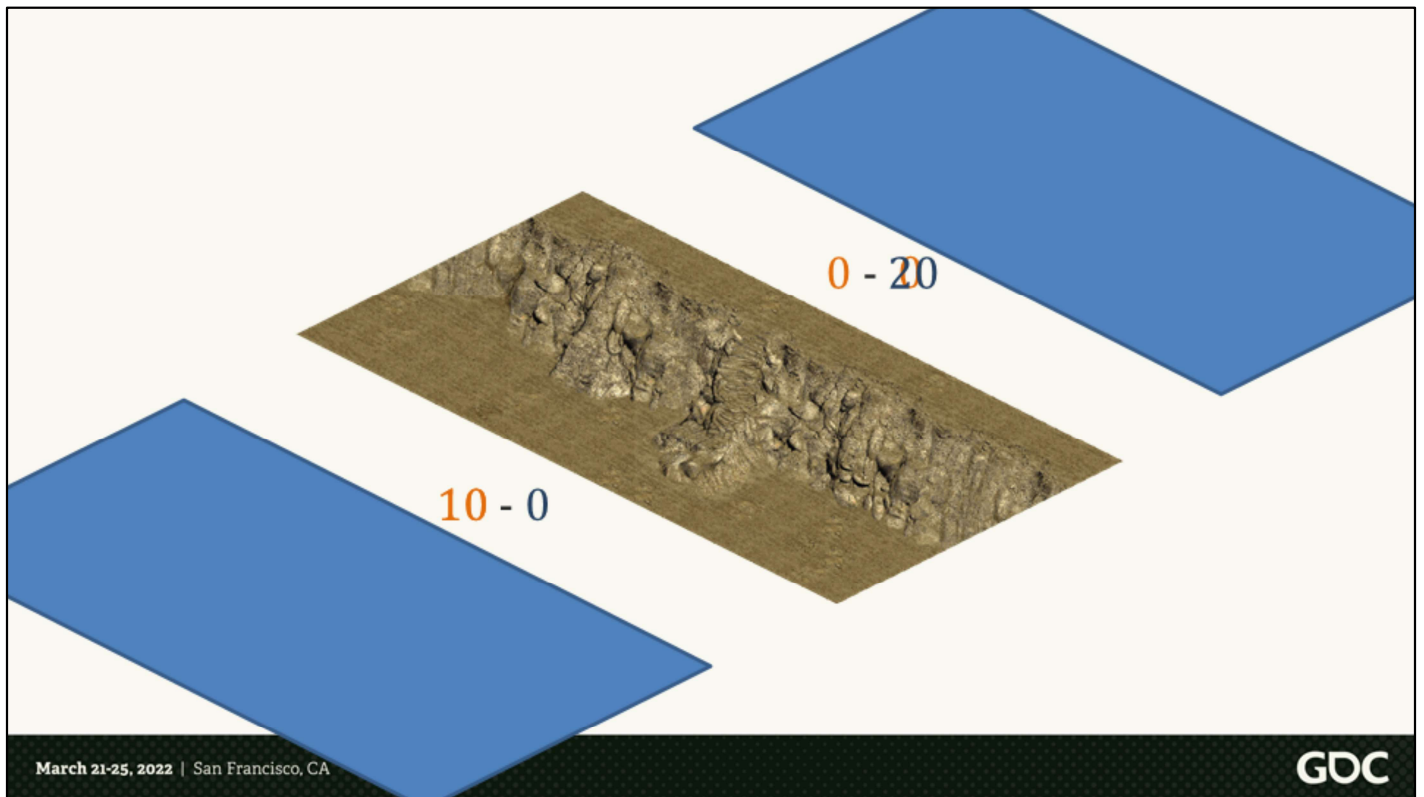
We look for presets in each cardinal direction around the one we want to place.



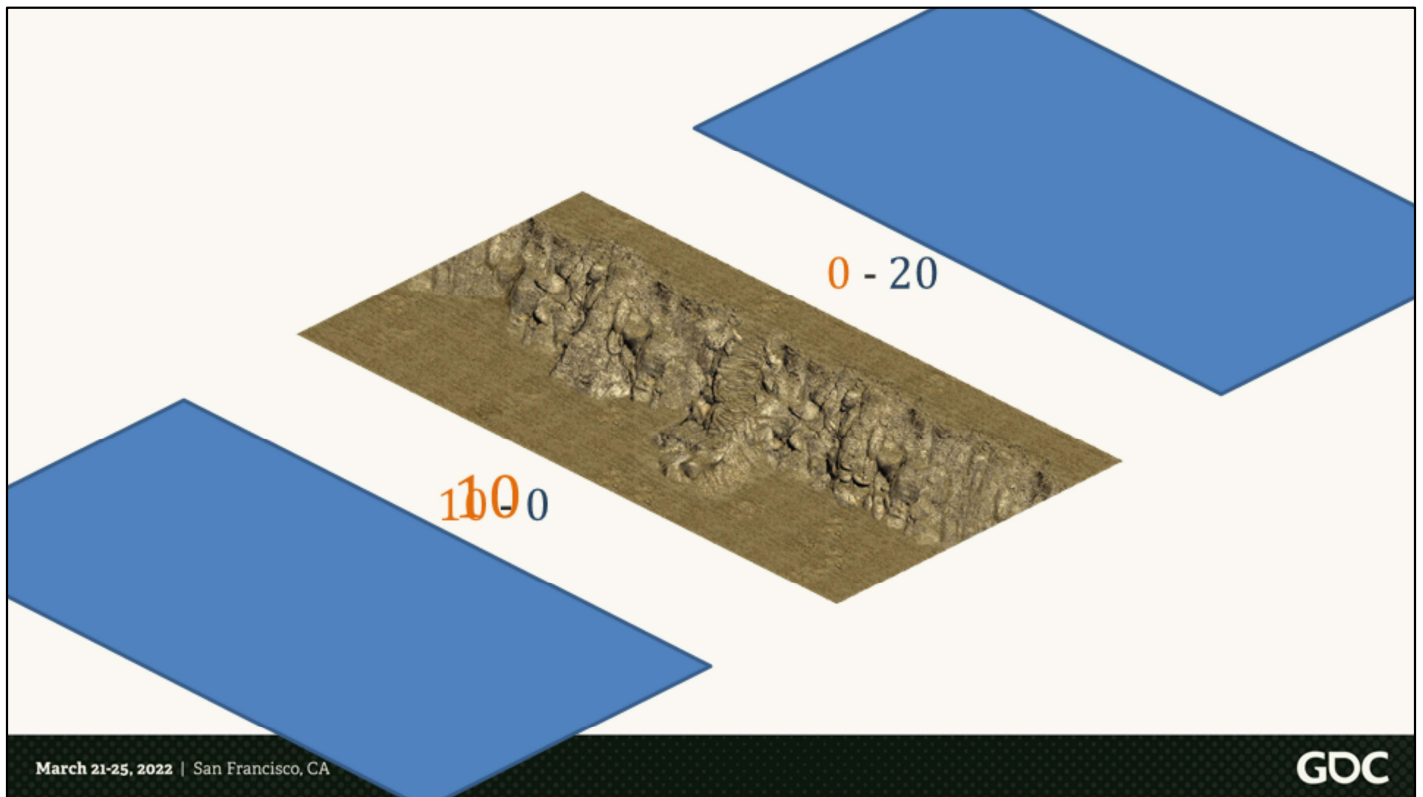
For this example, we can discard the northwest and southeast edges because they're always going to align with height.



We take the height markup for the border edges of orthogonal presets.



And subtract the height markup of the preset we're placing, and largest offset wins and is returned as the height offset we need to spawn the preset at.



In this example, that would be 10 feet, because the southwest border has a height differential of 10 feet. A valid question is: can any unresolvable height differences occur, such as trying to place a preset with a height change between two preexisting presets on the same level?



And the answer fortunately is, almost all of the time, no, because presets are placed outward from the player's position, so the most common case is only comparing heights against one edge, or two adjacent edges, and 99% of the time these situations are resolvable.

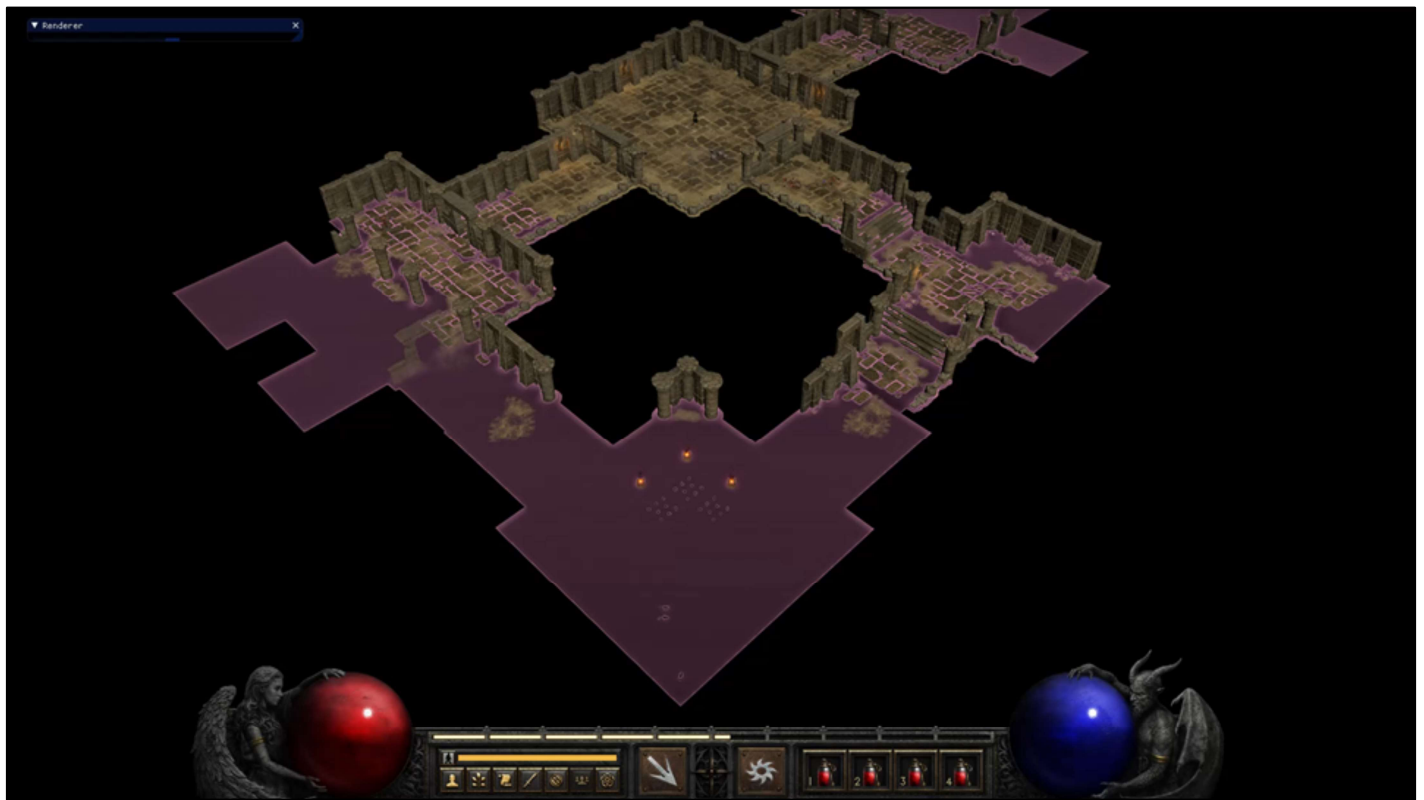


But Diablo II is also a game that doesn't make sense sometimes. We like to call this example The Loop of Death. You can see here that if one were to run around this dungeon loop clockwise or counterclockwise, you can technically travel down, or up, infinitely.

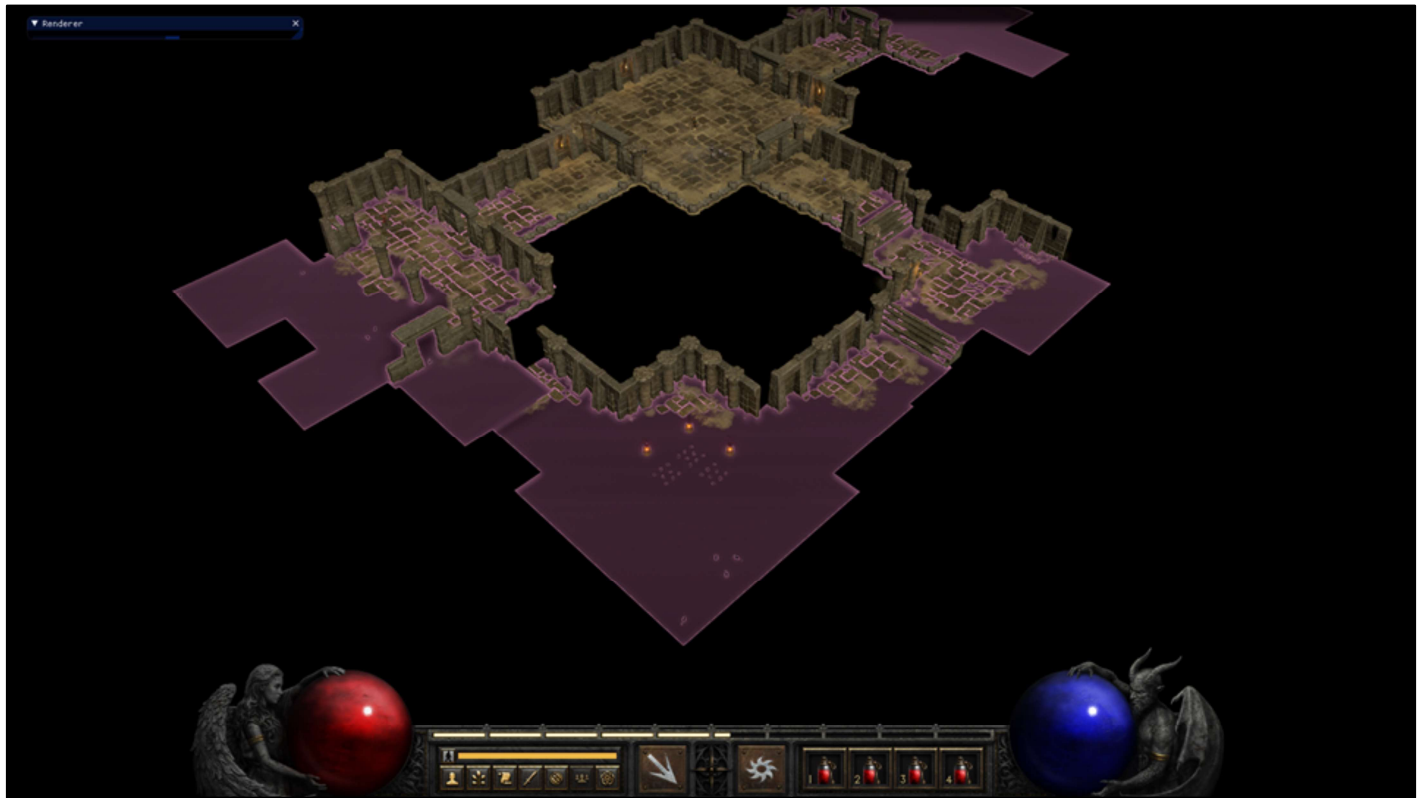
Preset Height

- Still-loaded presets can create unresolvable height scenarios
- Solution: constantly monitor and replace presets off-screen

This case was particularly difficult because the loop is small enough that all the relevant presets are still instantiated while traversing the whole thing. Our solution was simply to monitor and re-spawn presets off-screen if we detected that they no longer aligned at adjacent preset heights.



And you can see that in action in this video.

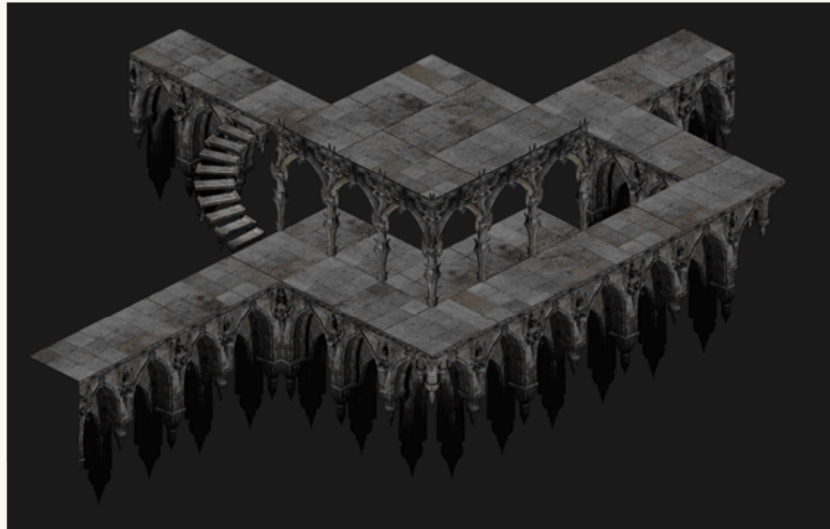


Now if we compare the start position with the end position, it looks like the camera moved. The camera didn't actually move, the entire level layout actually just moved downward, because overall the character travelled down, and off-camera we repositioned the presets to reflect this change in height.



Now if we compare the start position with the end position, it looks like the camera moved. The camera didn't actually move, the entire level layout actually just moved downward, because overall the character travelled down, and off-camera we repositioned the presets to reflect this change in height.

What about *that* level?



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Oh, and if you know Diablo II well you're probably wondering, how did we handle the Arcane Sanctuary? The whole concept of the space is geometry that's impossible in 3D. So how do we make something like this, real?



I hate to burst everyone's bubbles on this one, but the truth is that the walkable area, and even the staircase, is just flat. Even the original game forces an orthographic camera in this area, so we can use the same forced perspective tricks to give the illusion of impossible geometry. The staircase, while originally a real staircase, was flattened, to prevent a lighting pop when the character would step off the staircase onto the platform behind it at a different height.

Terrain

- We have a 3D environment!
- But wait! There's no floor.
- Can we include the floor in our presets?
- "One giant plane of controlled noise."

Ok great! At this point we had our 3D world coming together. But there's one pretty critical component missing – the floor.

Can we include the floor in our presets? Not really. The floor is going to need to be continuous, and presets can connect to a variety of other presets in any area of the game, and it sounds pretty horrendous to require our environment team to guarantee that all the edges of each preset are tileable with other preset edges, so that's out.

Instead we did something that I like to call one giant plane of controlled noise.



Terrain in the game is made up of a maximum of 10 layers, each of which has texture maps for albedo, normal, roughness, as well as data for grass, clutter, parallax properties, and more. Each layer also has parameters that control their influence in the terrain stack and thresholds against other layers in the stack.

Noise on the scale of the world coordinate system is used to calculate what terrain layer appears at a given position. The noise value is used to resolve the stack of all 10 terrain layers, with either one or a blend of two layers winning out at a given position. Art could manipulate the threshold values to get the terrain appearance and complexity that they wanted.



March 21-25, 2022 | San Francisco, CA #GDC22

GDC

This works great for general terrain but there are also cases where art needs to have specific features in specific locations. These features are specific to presets, so in a preset an artist can place stamps and decals that manipulate the terrain. Decals are specialized decals for terrain, and stamps are used to explicitly force certain layers to win out during the resolution of the layer stack.



There's also one more special case where we were able to leverage underlying data from the original game, and that's paths. Most prominent in act 1, paths are special tiles in the sprite graphics that are also procedurally placed in the world. By associating these tiles directly with chunks of generated terrain, we can automatically populate paths in the remaster without having to have art replicate them. All art needed to do is provide alpha masks that would affect the pattern of each path tile. Art could also manipulate the terrain layer which would be used to represent the path.

Terrain

- A preset has a floor with a special material
- The material will be textured according to its position in world space

Within preset files, an artist places a floor mesh with a terrain material, which indicates that mesh will be rendered with the virtual terrain textures. The pattern which appears on the floor ultimately depends on where in the world the preset is procedurally placed. It's a large procedural version of splat textures.

Terrain

- Resolving all 10 layers every frame for the whole floor is *expensive*
 - But that's where we started
 - >20ms on our workstations

Now, evaluating all 10 layers of the terrain stack at a given position is rather expensive. This is where we started, but it took something like 20ms of GPU on our development workstations, so of course this wasn't going to fly on consoles. Or any machine, really.

Optimize!

- We can cache visible terrain
 - Size is variable, 9k at most
 - Generated in chunks using compute

So it's time to optimize. The key observation here is that you're not going to see new terrain every frame. For a vast majority of the time, most of the terrain on the screen remains constant, so we can optimize terrain generation heavily by caching the result into a virtual texture. The composite size varies per platform, depending on the output resolution of the game, in a best attempt to get a 1:1 pixel density ratio. The highest quality we offer is a 9k composite texture (technically two for all PBR properties).

Generation of the texture is distributed into compute jobs operating on square chunks of the atlas. As you move, generated tiles would look like a sliding window across the atlas texture.



Optimize!

- There's variability in what's new
 - Sometimes it's 0 – you're standing still
 - Sometimes it's several – you're running
 - Sometimes it's nearly all of it – you're teleporting ☹️

This helps performance a lot, but our new problem is that there's a lot of variability in how many tiles need to be generated each frame. Sometimes it's 0, when you're just standing in one spot. Sometimes it's an entire row of tiles in the texture as you run around the world. And of course sometimes it's nearly all of it, if you're a sorceress and you're speedrunning the game ☹️

Optimize!

- Prioritize work
 - Highest priority are tiles in the camera frustum
 - Next, outside frustum
- Lastly, asynchronous compute

So, we optimized this even further by prioritizing what tiles we actually generate. Top priority are the tiles within the camera frustum. Again, we get to take advantage of our fixed camera, and we know automatically what tiles are visible on the screen. Any tiles that fall outside of the camera frustum we can amortize over several frames, unless they happen to come into view at which point we must generate them.

And lastly, we put terrain texture generation on asynchronous compute in order to hide the cost as much as possible.



Where are we?

- Make the illusion of 3D real.
- Make it look like everyone remembered.
- We need a 3D engine.
- We need a toolchain.
- We need the right rendering technology.

At this point, we've established our 3D space, we've set the art team off and running with tools to make the content, and we've solved our issues with height changes in the game (mostly, there were still a lot of bugs). We still have another large undertaking ahead of us - how do we recreate what people imagined when they were playing this game decades ago? Our brains can do some amazing things, and players turned those pixels into some pretty impressive imagery.

More Talks!

- *Diablo II: Resurrected*: The Art Direction and Pipelines for Resurrecting Evil
 - Dustin King
 - Thursday, 4-5pm, Room 2006, West Hall
- An Overview of the *Diablo II: Resurrected* Renderer
 - Kevin Todisco
 - Wednesday, 3:30-...on the vault!

Now, it's definitely worth me mentioning that a ton of work went into the art direction for the game to pull off that visual trick of recreating what was in your mind's eye back in the day, and if you'd like to hear about that, go see my colleague Dustin tomorrow in the West Hall.

And on the technical side, you can hop in your time machine and go back approximately 2 hours and 15 minutes to see me talk about the renderer in detail in this very room... or see it when it hits the vault. I'm only going to talk about a few very niche things we did with the renderer here so if you want the full story on that, I definitely recommend checking this out.

Rendering Technology Impact

- Largest investment
- Fixed camera to the rescue.
 - No long-distance vistas
 - No sky
 - Fixed terrain visibility

The rendering technology we would choose would play a big role in our success, not just from a time perspective because we were building everything from the ground up; but also, some key details were necessary to really nail the same visual feeling a sprite game evokes.

There was one really important detail for us to work with, which was that the camera is fixed. This meant we could take a lot of shortcuts and solve a lot of problems by leaning into this. It also cuts down on the number of systems and performance situations we need to worry about. We don't have to worry about long vistas, and therefore don't need to create any kind of distance imposter rendering systems – though we would still need LoDs and rate limiting for platform scaling. We don't need to render sky – like, at all. You can't see it. And, as I talked about earlier, we know exactly how much terrain we can see at any given time, and we used that to our advantage.

Visual Goals

- Physically-based, realistic visuals
- Exactly what people thought they saw in the year 2000
- So, start with a physically-based renderer...

Art's ask was physically-based, realistic visuals. Goal-wise, we want to recreate exactly what you thought you were looking at in the year 2000. So, we start by building a physically-based renderer and see what we get...



So, ignoring some of the artifacts here, this is what you get, when you take the act 2 desert and light it for the daytime with universal lighting. It's probably clear that there are some monsters right in front of me. But what about this body over here? Is there loot around it? Is it obvious that I can interact with it? Oh, I bet you didn't notice that there's a monster literally right next to me.

Key Element Visibility

- With universal lighting, it's really hard to see certain things
 - Characters
 - Items on the ground



If we look at the sprite game, sprites in motion like monsters stand out from the environments a lot better. It's almost a little bit of that effect from old cartoons where the art style of the secret doorway was different than the rest of the wall, so you kinda knew something would happen with it.

Key Element Visibility

- With universal lighting, it's really hard to see certain things
 - Characters
 - Items on the ground
- These elements are critical to gameplay.
 - Realism needs to step aside.

Either way, characters and items on the ground, and interactable things are elements critical to gameplay, so the realism needs to step aside, and we need a way to bring this readability into the remaster.



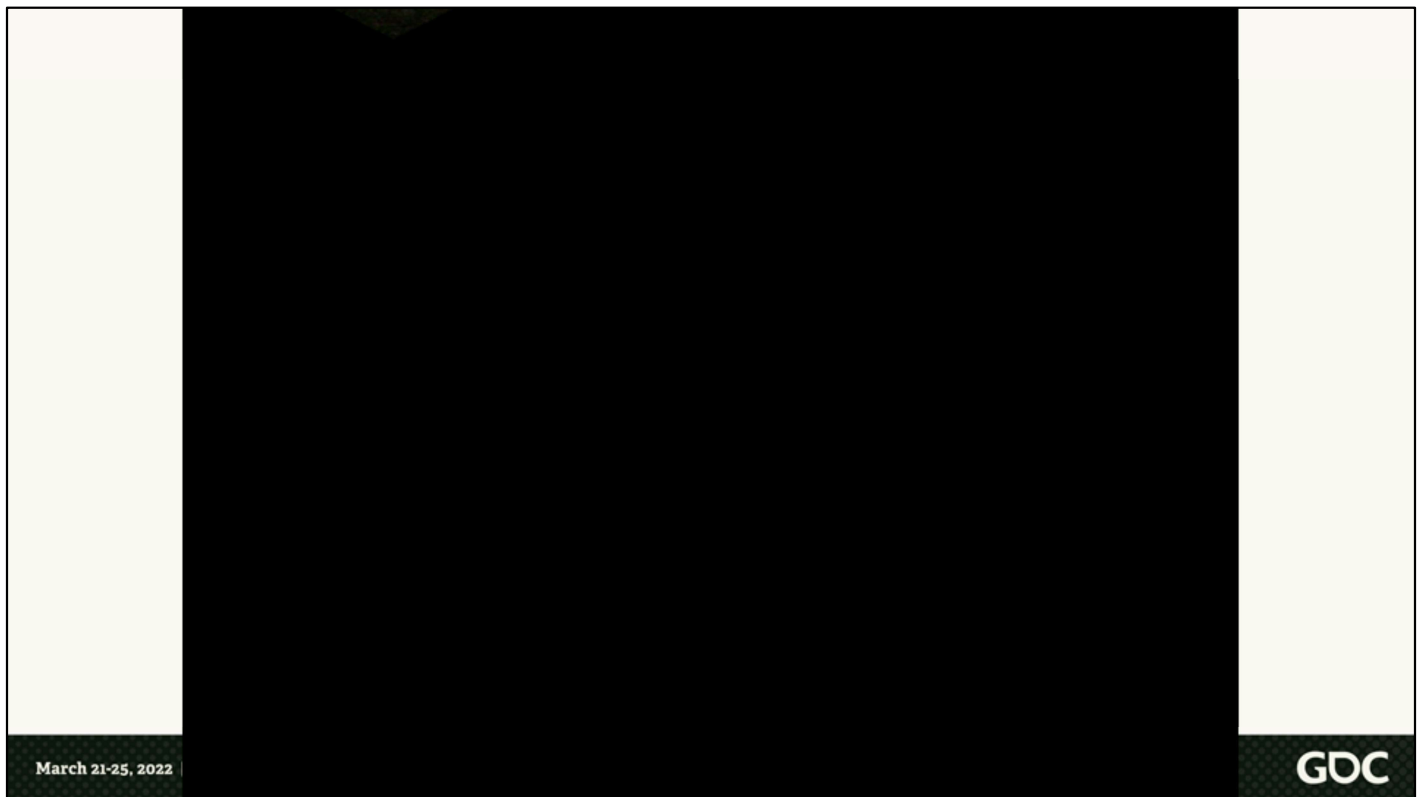
Our solution was to give players, monsters, and even items their own unique lighting rigs -- not per instance mind you, but per type. Players are lit by three directional lights and monsters and items are lit by two. Art could control the direction, intensity, color, and diffuse and specular contributions of the lights independently from the environment. The properties of these rigs are primarily controlled by the time of day system, as are the environment lighting properties, but can also be overridden by trigger volumes throughout the world, such as walking into an interior space from an exterior space.



Here you can see the lighting rig for all monsters being modified.



And, lastly, there were global controls over the brightness of characters and items that were applied in a pseudo-post-processing step at the end of the lighting shader. It worked just by modifying the value in the hue/saturation/value representation of the shaded pixel color.



Something else that a sprite-based game has that a 3D game struggles with is explicit control over layering and ordering of the objects in the scene. Diablo II always renders elements in an order that visually makes sense – like in this animation that I showed earlier. Floor, then shadows, then walls, characters, items, then weather, then UI. If some gameplay element must appear in front of another, it's fairly easy to do so.



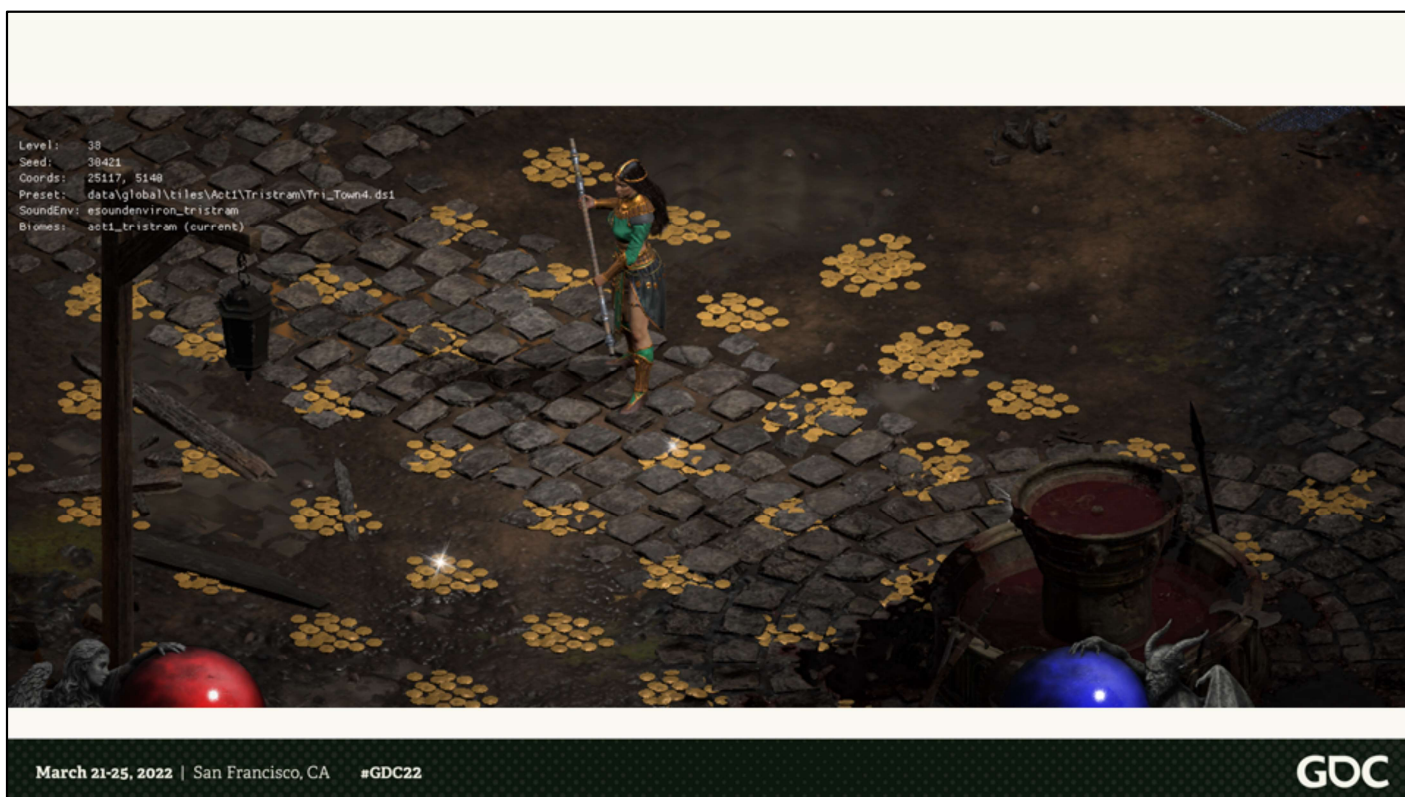
As an example, a character can stand on top of anything on the ground and draw over it completely.



But in 3D, everything is just going to clip together.



Some of the key issues we had were problems with characters and items clipping with the ground or non-gameplay elements on the ground, which in the best case would make your character's feet disappear but in the worst case would make it impossible to see the item that just dropped on the ground.



Like gold.



Or sweet loot.



Even bulky armor pieces could sink into the ground a little bit.

Draw Ordering

- Draw important objects like characters and items first.
 - You must be able to see things that drop on the ground!
- Set a stencil bit for these draws.
- Test that stencil bit when drawing decorators.

To solve this, we leveraged the stencil buffer during our depth prepass, first drawing out characters, items, and any other objects that needed to appear above the terrain or other terrain elements. We provided the art team a means to tag models as ones considered to be part of the floor, such as these pillows in the harem.

These objects would draw later in the depth prepass but fail the stencil test anywhere that a character or item had drawn, and therefore not render there at all.



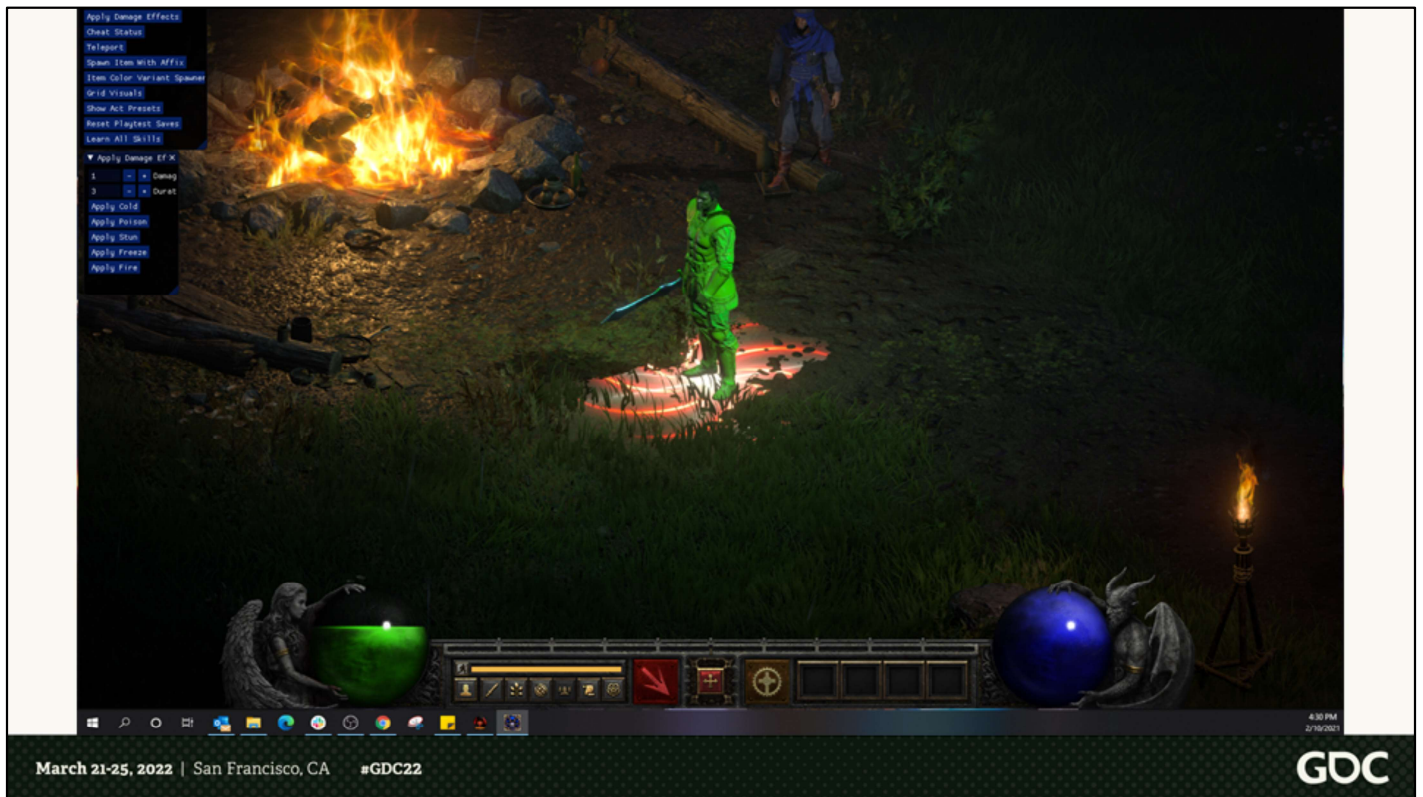
So we can go from feet sinking into harem pillows.



To a character fully visible on top of them. Normally in 3D this makes no sense. Or you might put collision on the pillows and stand on top of them. But our 3D elements are all being positioned by a 2D engine that doesn't really care that pillows are not flat like paper.



There was one case that this would not work for though, and that was ethereal items. These items render partially transparent, and that precludes them from completely obscuring whatever is behind them, meaning we can't prevent rendering whatever is meant to blend with them. They're drawn later in the frame, by the time the terrain is already drawn. This is something we never solved and counted on being a rare enough case that it wouldn't be a huge issue. It remains an open problem.



We also had issues with auras. Since we didn't have a projected decal system, our auras are planar effects parallel to the ground that were originally placed directly at ground level. Whenever there would be undulation in the ground or any kind of grass or foliage at ground level, the auras would clip through them.



For this, we took advantage of the fixed camera angle. We asked the effect artists to move the origin point for the auras off the ground toward the camera, and we rendered them testing against the same stencil that characters and items would draw with. By moving the effect along the camera vector, its position on screen did not change, and by ensuring a character always draws over it, it guarantees that the aura still appears to be at the character's feet, without clipping with grass and foliage.



Speaking of grass... it tends to get in the way of items as well. For this, we cut a two-for-one deal with visual fidelity that serviced important gameplay elements. We render out force maps on grass that are affected by characters, items, and objects which are then used during grass rendering to affect the bend of the grass blades. Blades that fall in the same location as a character's feet, items, or objects placed on the ground will essentially flatten out and prevent clipping with the object. Plus it's a cool detail to run around and see the grass deform.

Darkness

- Major component of Diablo II aesthetic
 - Danger, unknown
- Realistic lighting doesn't lend itself well
 - Ambient light is aptly named
 - Big emphasis on GI today
- Light is information

Now really the biggest thing in Diablo II is the darkness, and this is where a lot of our attention went. As part of our physically based renderer, we were using image-based lighting as a global illumination solution. The real issue with modern rendering is that the simulation of bounce lighting in enclosed spaces will mean that even a single light source in that space will provide *some* amount of illumination to the rest of the surrounding environment.



In Diablo II, the darkness in dungeons beyond the player's light is total. Unless a monster has a light as well, you will not be able to see anything beyond that light falloff. Light and sight give information. This meant that in order to maintain this behavior in Resurrected, we needed to break the physicality of typical light.



In the absence of something to suppress light in those same areas, you get this. It becomes wayyy more advantageous to play the game in Resurrected mode – you can see everything!

This is something that no other game out there really needs, so it was something convenient for us to solve in engineering without art needing to work around some existing engine behavior.



The system we came up with we called global attenuation. An extra rendering pass uses light data and depth information to render out a fullscreen buffer that describes the influence and extent of any punctual light source in the scene.



This global attenuation mask is multiplied over lighting in order to suppress the influence of ambient light in those areas that were not lit by a punctual light source, meaning that if any area had *no* light sources reaching it, it would be completely dark.

Global Attenuation

- Nearly every lighting element is affected by global attenuation.
 - Emissive materials
 - Lighting rigs
 - Image based lighting

Nearly every secondary lighting element in the game is affected by global attenuation in order to achieve these pitch dark areas.



In the case of character lighting rigs, it turns full lit monsters like this...

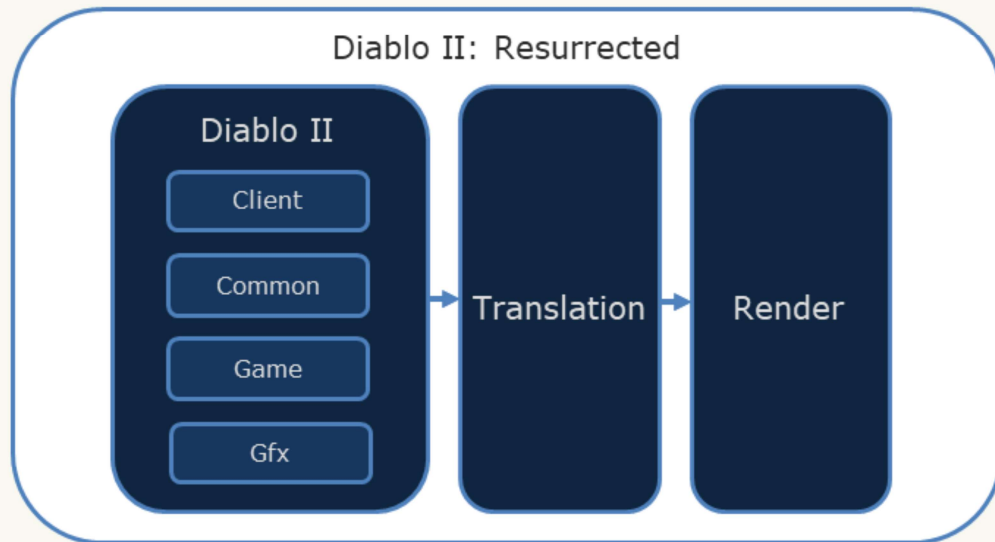


Into much more intimidating looking monsters emerging from the darkness.



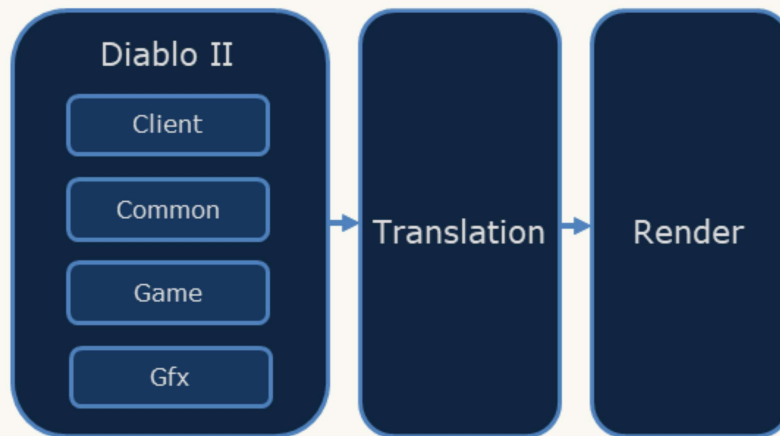
And it's a pretty close match to the original game, so we were really happy with this.

Legacy Toggle



There's one more thing to talk about that I think is worth mentioning. Remember this diagram of the software that I showed earlier? The software layout can also be read as a flow chart for how a frame of the game unfolds.

Legacy Toggle



The frame starts with all the original game logic, sprite rendering included, then goes on to translating gameplay elements into 3D, and finally renders the 3D scene.



Under the hood at all times, the original game is still rendering since the cost is negligible (ok, well, there are *some* shortcuts we take but for the most part we can consider it negligible).



And then the 3D graphics just draw right over it. Which gives us this unique opportunity to decide at the end of the frame which version to show you. And switching between them is as simple...



... as a fade.

```
float3 result = lerp(sdColor, hdColor, fadeValue);
```

This is the code that does that. Now, I show this line because it's one line of code that creates the effect, combined with the opportunistic software structure that made it possible. And, when I advocated to have this feature in the game, I knew that it would be this nice little addition but I couldn't imagine just what the reaction would be. When we first put this in, the reaction internally was pure joy. I laughed, because when you think about it as an engineer, this is so simple. I thought to myself, of the many more technically challenging things we've put into this engine throughout this project, it's a single lerp function that's evoking one of the most excitable reactions from the entire team.

Pressing 'G' is just so mindblowing... I created a L80+ character just so I can run around Normal and spam 'G' and not worry about dying. The atmosphere in this game is just incredible. What's your favorite area in D2R?

I just love it because the remastered graphics are exactly what my MIND remembers the game looking like, even though deep down I know that's not possible. It's amazing toggling back and forth lol.

It's also quite fun to compare the two styles at different points. It allows players to see how Blizzard has updated the classic since its original release. The Lord of Terror himself looks particularly fearsome in the new art style.

Instant Swap With Diablo II: Resurrected Graphics Is Nostalgic Magic

If you haven't already, toggle. Jesus. Wow. It's amazing how when you were 15 this game seemed at peak graphics and only now does the remaster match your 15 year old memories. Toggling to legacy is like...going from a Tesla Model S to an '84 Honda.

And it wasn't just the team. The whole internet found this feature to be incredibly fun. Kind internet strangers, streamers, and journalists alike got a lot of mileage out of it, at times being pulled away from playing the game and just walking around, zooming in, and switching back and forth between old and new graphics to compare what the modern interpretation looked like.

Courtesy of: Reddit, <https://www.dexerto.com/diablo/diablo-2-resurrected-how-to-swap-between-modern-legacy-graphics-1646984/>
<https://www.gameinformer.com/2021/04/09/instant-swap-with-diablo-ii-resurrected-graphics-is-nostalgic-magic> , Blizzard Forums

Legacy Toggle

- Small feature, big impact
- Promoted the nostalgia
- Backed up our claims about the engine

The lesson from this being even the smallest features can have the biggest impact. I also think that another reason this feature was so successful was the way that it helped promote the nostalgia that the game attempts to evoke. And, in the same vein, it backs up our claim about the original engine running under the hood. The legacy toggle is always there as a reminder that it's the exact same game that you played 20 years ago, but with a facelift.

What did we learn?

- Maybe don't build an engine from scratch
- Leverage the shortcuts you have
 - A fixed camera is a big advantage
- Breaking the rules is OK
- The Devil is in the details

So coming out of this, what did we learn? Honestly, maybe don't try to build an engine from scratch. It's a huge undertaking, larger than it always first appears, and while it can offer you lots of opportunity to tailor your solution specifically to the game you're making, it's easy to take for granted the swaths of foundational technology required to make a game that you'll be missing up front.

Second, always be on the lookout for shortcuts that you have when making any game technology. In our case, our most important advantage was the fixed gameplay camera. We took advantage of this in different ways from terrain optimization, to skipping sky rendering entirely, and solving many of the draw ordering issues that are difficult to pull off when recreating a sprite game.


Third, breaking the rules is ok. We had to depart from strictly physically based lighting in many instances, creating an amalgamation of lighting approaches in order to service gameplay and bring the remaster closer to the look and feel of the original.

And lastly, the devil is in the details. Little things like simulating sprite drawing and features like the legacy toggle are great examples of how the small details can really improve the quality of a remaster, or any title for that matter.

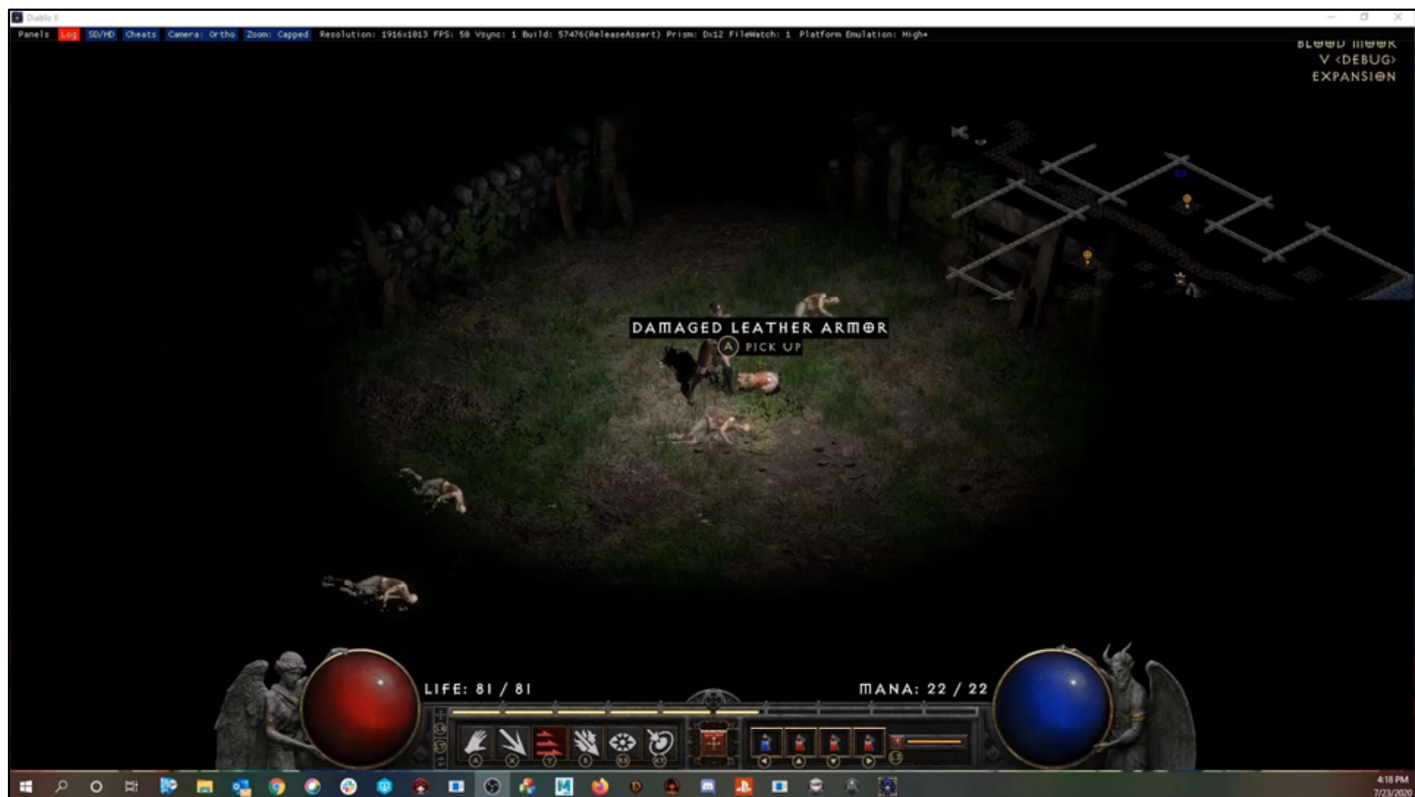
Special Thanks

- The entire D2R graphics team
 - Chad Layton
 - Ace Stapp
 - Joel Peters
 - Jon Lee
 - Gustavo Samour
 - Nish Sundharesan
 - Anushka Nair
- Iron Galaxy
- The QA team
- The entire D2R team
- Persistent Studios
- Michael Bukowski
- Andreas Fredriksson

Questions?

 @kevintodisco
ktodisco@blizzard.com

We're hiring!
<https://careers.blizzard.com/global/en>
<https://careers.blizzard.com/global/en/albany>





Diablo II / D2-49289

[Act 3] docktown_food lobster model is incorrect species of lobster

Steps:

1. Load a character in build 62211
2. Teleport to Act 3 town
 1. goto 75
3. Walk over to the market area and find lobsters
4. Observe issue

Expected Results:

- Due to Act 3 taking place in a southern, tropical environment, lobsters should be Langustas, or Spiny/Rock

Actual Results:

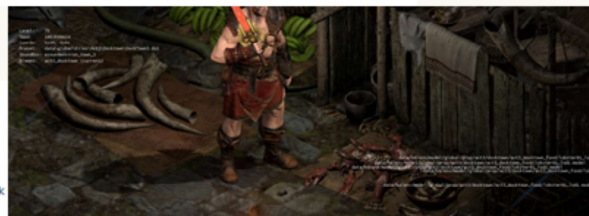
- Lobsters are of the northern, or "True Lobster" type with claws

Additional Info:

- This lobster type does not make sense due to the location of Act 3. In order for a merchant in the town to be able to offer a "True Lobster", they would have to have it shipped in from a more northern region (like from the area around Mount Arreat)

However, the presence of lobster traps nearby suggest that the merchants are catching the lobsters themselves, which would mean that they *should* have the spiny lobster variety.


See attached images of lobster distribution and the Diablo 2 world map.



Types Of Lobster

The greatest bug ever.

Questions?

 @kevintodisco
ktodisco@blizzard.com

We're hiring!
<https://careers.blizzard.com/global/en>
<https://careers.blizzard.com/global/en/albany>

Appendix

Stats

- Lines of code written for 3D layer: 135,224
 - 102k for D2Render, 20k for Translation, 5k for D2Entity, 7k for D2Glue
- Lines of shader code written: 31,132
- Total # shader files: 229
 - 141 HLSL and 88 HLSLI
- Total # HD textures: 29,990
- Total size of all source textures: 322.7 GB
- Total #Granny files: 29,708
- Total # VFX files: 1,851

2D *and* 3D in Harmony

- Having a pixel-perfect formula for converting a 2D coordinate to 3D provided an early-development benefit.
- We were approaching a milestone. Lots of content wasn't in yet.
 - Characters, animations, etc.
- One of the artists says "wouldn't it be great if we could just reuse the sprites for these missing things"
 - Programmer: "Wait a minute, we can totally do that"

D3D12: F12, TPrntScrn to capture Frame: 944, 46.96ms (40.56 .. 108.24) (21. FPS)
0 Captures saved.

