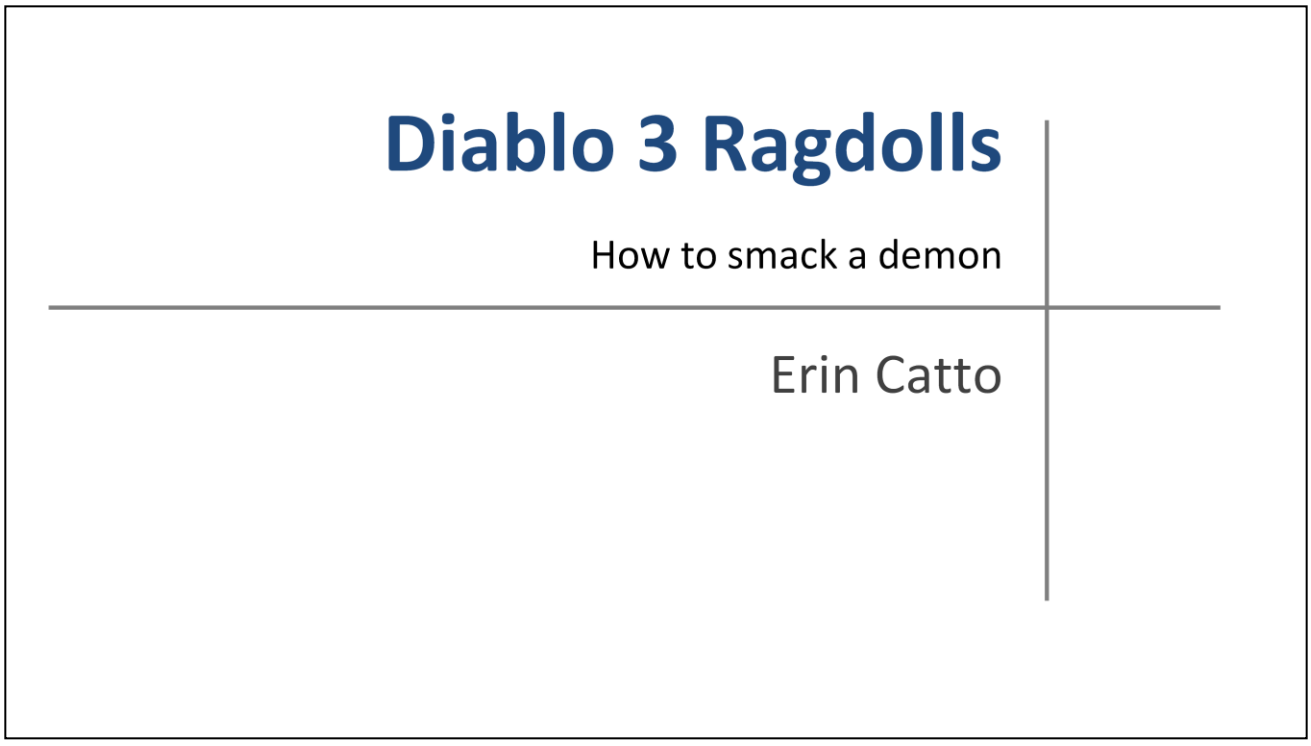


<title page>

Hello everyone!

My name is Erin Catto and I want to thank you for coming to my tutorial.

Today I'm going to discuss my work on the Diablo 3 ragdoll system. I learned a lot working on this project and I hope you find this information useful in your own projects. And as I promised, you will learn how to smack a demon.

First let me give you a little background about myself.

Who am I?

Box2D

<who?>

My first job in the game industry was writing the physics engine for Tomb Raider: Legend at Crystal Dynamics. The engine was used to create physics puzzles, ragdolls, and vehicles. The same engine lives on today in Lara Croft: Guardian of Light and Deus Ex 3.

In my spare time I have been working on Box2D, a free open source physics engine. Box2D is used in Crayon Physics, Limbo, and Angry Birds.

So I got this job at Blizzard …

<setting>

So after shipping Tomb Raider, I got an amazing opportunity to go work at Blizzard's console division. That same division had been working on StarCraft Ghost. When I arrived at the console division, Starcraft Ghost had been cancelled and they were brain-storming for new game ideas. I didn't pay too much attention to that because I had been put in charge of developing a new physics engine for the team. I was excited to work on the Xbox 360 and the PS3. And I wanted to improve on what I had done at Crystal.

Well two weeks later the console division was shut down. I was told to go home and wait for a call to see about other positions at Blizzard.

I ended up as the physics programmer on Diablo 3

<role>

As it happened, I was rescued by Tyrael!

I must confess that I was a Diablo 2 addict. I actually uninstalled it once and threw away the disks because it was taking too much of my time. Nevertheless, I have many fond memories of the game.

I was therefore incredibly excited to become a part of the Diablo 3 team. I wanted to help make the best Diablo game ever!

I wanted to give Diablo the most awesome ragdolls ever!

<goal>

So when I started thinking about bringing physics to Diablo, one of my first ideas was ragdolls. I wanted ragdolls on every creature. I wanted it to be easy for artists to create all kinds of over the top ragdoll effects.

A good ragdoll system has a lot to offer.

From a design point of view, ragdolls make the world feel more interactive and make the player feel more powerful.

From an artistic point of view, ragdolls allow for lots of death variation and reduce the animator workload.

From a programming point of view, ragdolls improve the visual quality by making the corpses conform to the environment. Ragdoll physics prevent corpses from hanging off a cliff.

Ragdolls == physics engine + other stuff

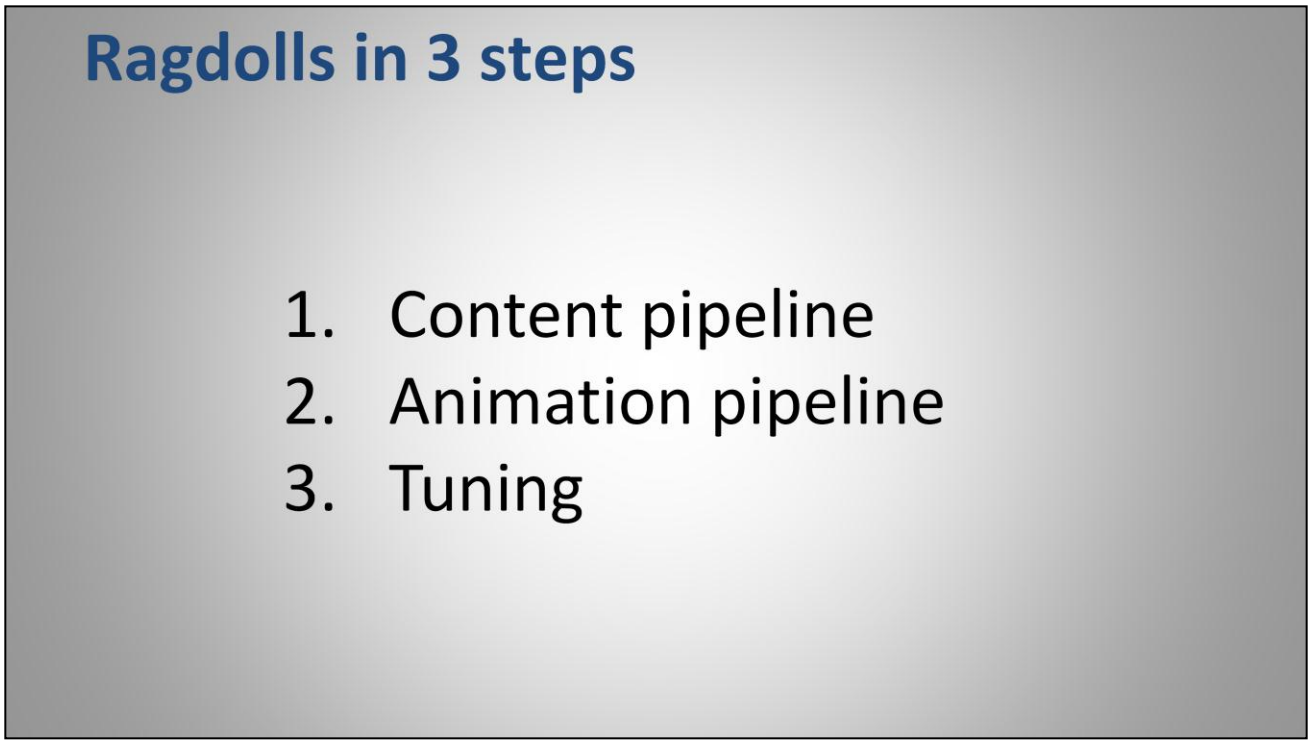
... what other stuff?

<challenge>

I wrote a ragdoll system for Diablo 3 on top of the Havok physics engine. We later switched to an in-house physics engine called Domino. Now I love physics engines just as much as the next guy (well maybe more so), but today I'm mainly going to talk about my work in making ragdolls a core feature of Diablo. Most of that is code written outside of the physics engine.

One thing to keep in mind is that a physics engine doesn't know a ragdoll is a ragdoll. The physics engine just sees some rigid bodies, collision shapes, and joints. The physics engine doesn't know how the ragdoll is authored, how it hooks into the animation system, or how it gets whacked by an axe. The physics engine doesn't keep your game from embedding a ragdoll in a wall, or make sure your ragdolls are stable and look good. These aspects are usually handled by a programmer on the game team, not someone from the physics engine company.

Sometimes this work is done by a physics programmer, but often this work is done by a generalist programmer who is not familiar with the inner workings of the physics engine. Some aspects of implementing a ragdoll system are straight forward, but others are not always so obvious. And navigating the pitfalls of a physics engine is not always easy or intuitive.



<call to action>

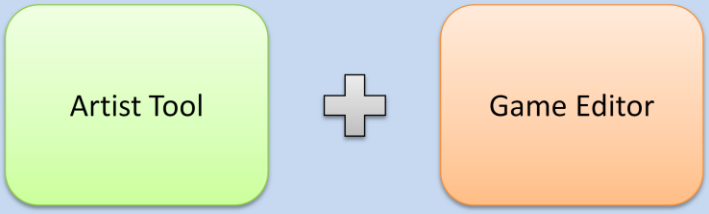
There were three main areas I had to tackle to build the Diablo 3 ragdoll system:

First, I needed a solid content pipeline so that artists and riggers can author ragdolls on a large scale

Second, I needed to create some ragdoll glue to connect the ragdoll to the animated actor

Third, I needed to develop some tools and know how to make ragdolls stable, look good, and add flavor.

We need a content pipeline to author and tune ragdolls



<key point>

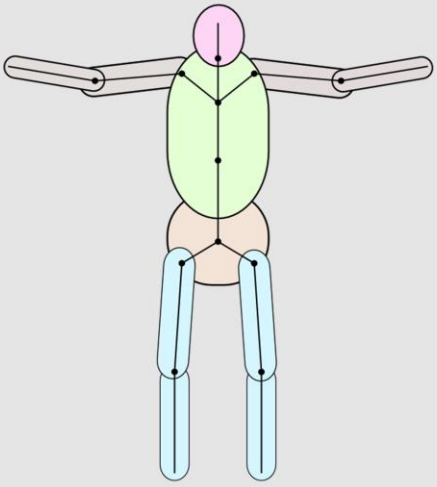
Now let's talk about the content pipeline. The Diablo ragdoll content pipeline is a wedding of the artist's tool and the game editor.

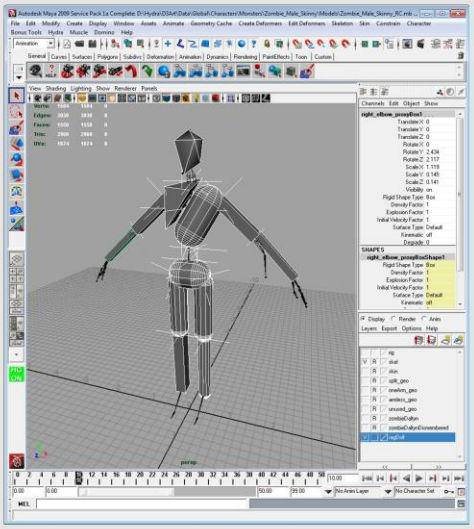
I wanted ragdoll geometry to be defined in the artist's tool. This centralizes most of the geometric data and the artist's tool is usually well suited to geometric construction.

It is not easy to get the ragdoll simulation running the artist's tool. Sure, we could probably integrate the physics engine into the art tool, but much of ragdoll behavior is tied to game logic. So instead, our artists test and tune their ragdolls in our game editor.

This is a fairly seamless process. There are no text files to be edited and much of the authoring is visual.

A ragdoll is a collection of collision shapes connected to bones





<explanation>

The ragdoll geometry is created by a technical artist in Maya.

We start with a standard character skeleton. The ragdoll is defined in the so-called *bind pose*. A skeleton's bind pose fits the original character mesh that was created by a modeler. The bind pose is the pose where the mesh is bound to the skeleton in a process called skinning.

We attach collision shapes to the skeleton bones. Each bone has zero or more collision shapes. These implicitly define the rigid bodies. Physics joints are represented as locator transforms that connect two bones.

With this framework, we can author all the physics objects in the game as ragdolls.

Video: Zombie Fest

Here are your standard death ragdolls.

Video: Chandelier

Here is a destructible chandelier. The entire chandelier is one model with a single skeleton. You'll have fun dropping these on top of zombies in Diablo3.

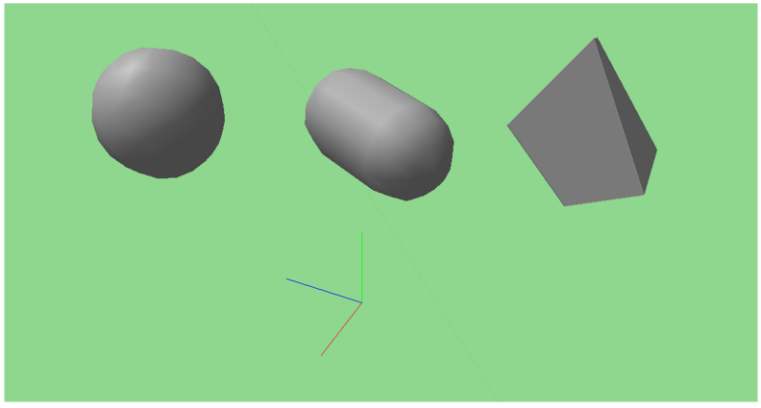
Video: Wagon

Even a destructible wagon is a ragdoll. So a ragdoll in Diablo is not just a character. This simplifies our authoring path and our code. Also, by having all physics props as ragdolls we keep draw calls low since the moving parts are just bones, not separate actors.

Video: Barrel

This barrel mixes destruction with a character ragdoll. But it is still one draw call.

We support spheres, capsules, and convex hulls



For collision shapes, we support spheres, capsules, and convex hulls. There is no special case for boxes.

We have a tool that can shrink wrap a vertex cloud with a simplified convex hull. The rigger can select a set of vertices and press a button to generate an efficient convex hull.

We try to use spheres and capsules where appropriate because they are cheaper than convex hulls. We have rolling resistance to prevent spheres and capsules from rolling forever, even on slopes.

Domino Demos

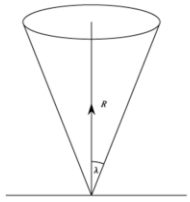
- Rolling resistance
- Convex hull simplification

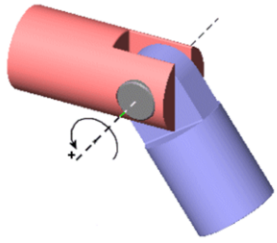
Rolling resistance improves behavior and helps bodies get to sleep faster.

Convex hull simplification improves artist productivity and improves simulation quality. Broad polygonal faces improves quality by eliminating a teetering effect that sometimes happens on shallow edges.

Physics joints connect two bones

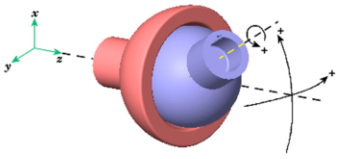
Cone Joint





Revolute Joint

Spherical Joint



Weld Joint

The technical artist connects bones with physics joints. The physics joints do not necessarily line up with the bones and we support branch crossing loop joints.

We use cone, revolute, spherical, and weld joints.

Cone joint: like a shoulder with cone and twist angle limits

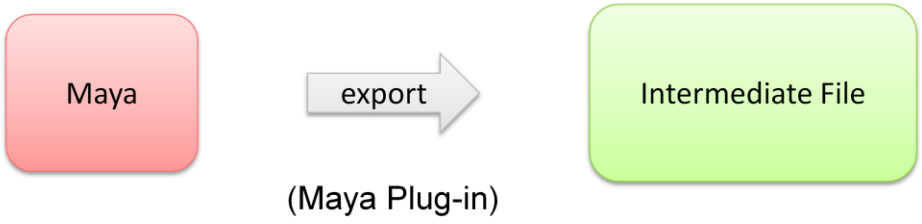
Revolute joint: like an elbow with a single axis of rotation and angular limits

Spherical joint: a pure point to point constraints with no rotation restriction, useful for chandeliers where rotation is naturally limited by collision.

Weld joint: locks two bodies together, useful for some advanced effects you'll see later.

We author physics joints using a Maya construct called a locator, which is basically a transform with position and orientation. So they are just coordinate axes. We have conventions to relate locator axes to joint axes. For example, the z-axis is the hinge axis for a revolute joint.

The ragdoll data is exported with the model



The ragdoll collision shapes and joints are exported with the model into a raw intermediate file. The export is handled by a Maya plug-in written in C++.

I highly recommend exporting data raw from your authoring tools, with minimal pre-processing. This allows the run-time format to change without requiring re-exports from the art tool.

Ragdoll data is imported by the game editor (called *Axe*)

<explanation>

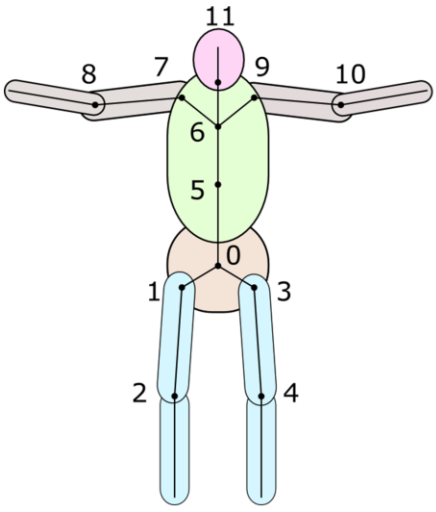
We import the intermediate file into our game editor, called Axe. During this process the skeleton is built along with the associated ragdoll data. The editor is aware of versions, so if the model version changes, the editor simply re-imports the intermediate file. No re-export from Maya is needed. The model is then serialized into an optimized binary format that can be read in-place into the game with simple pointer fix ups.

This run-time data is the game's representation of the ragdoll. We still need to instantiate the physics engine objects according to game logic. The process is quick and is mostly gated by the cost of inserting collision shapes into the physics engine's broad-phase.

Bones are sorted in parent-child order

```
struct Bone
{
   int parent;
   int child;
   int sibling;
};

#define NULL_BONE (-1)
```



Here is how we represent the skeleton. A bone is just a node in a hierarchy. Each bone should know it's parent bone with the special case that the root bone's parent is null. A bone also has a linked list of children. A ssibling index acts as the next pointer in the child list.

You will save yourself a lot of headache and improve performance by keeping bones in a sorted array. The array should be sorted so that parents always precede their children. This makes it easy to compute the bone transforms in model space. Often you don't even need to sort the bones because the art tool keeps them sorted.

I've worked on game engines that sort the bones based on other criteria, such as the name. Don't do that.

Here's how we compute model space bone transforms

```
// MS: model space
// PS: parent space
for (int i = 0; i < boneCount; ++i)
{
    int parent = bones[i].parent;
    if (parent != NULL_BONE)
        transformMS[i] = transformMS[parent] * transformPS[i];
    else
        transformMS[i] = transformPS[i];
}
```

Animation data is interpolated and blended in parent space. We then convert the parent space transforms into model space transforms so we can render the model.

When the bones are sorted parent first, we can simply loop through all the bones to compute the model space transforms. Without proper sorting, we would need a recursive algorithm to update the skeleton.

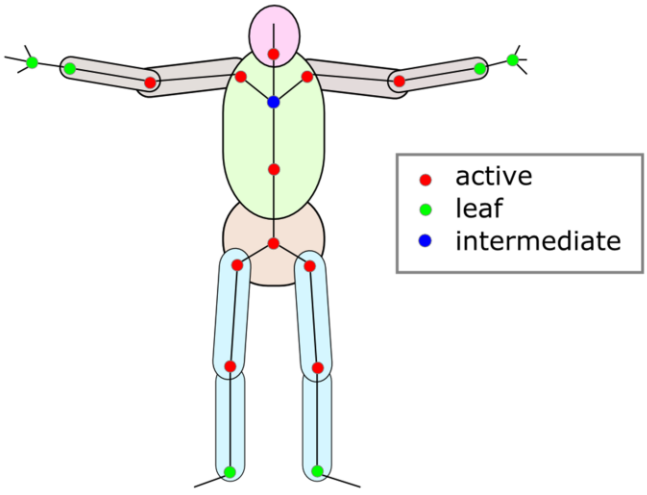
I also find that keeping the bones sorted makes it easier to synchronize the skeleton with the ragdoll.

Bones with shapes are bodies, the rest are leaf or intermediate bones

```
#define LEAF (-1)
#define INTERMEDIATE (-2)

int boneToBodyMap[BONE_COUNT];

struct RagdollBody
{
    int boneIndex;
    int parentBody;
    RigidBody* body;
};
```



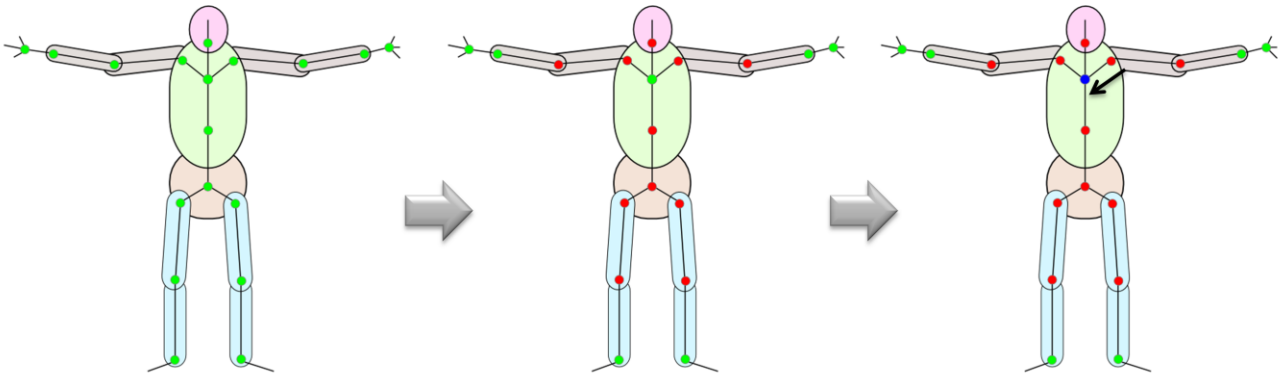
A typical character skeleton may have dozens of bones. There are usually far more bones than can be efficiently simulated with rigid bodies. Therefore, we usually only have rigid bodies on a sub-set of the bones.

This creates a situation where there are three kinds of bones: active bones, leaf bones, and intermediate bones. An active bone is attached to a rigid body. A leaf bone has no body and no descendent bones with bodies. An intermediate bone has no body, but has a descendent bone with a body. In other words, an intermediate bone is an inactive bone stuck between active bones.

The run-time ragdoll system needs a map from bone to body. That map has holes, but the holes are important.

To keep track of things I also have a RagdollBody structure that keeps the bone index, the index of the parent body, and the rigid body from the physics engine.

Building the bone to body map



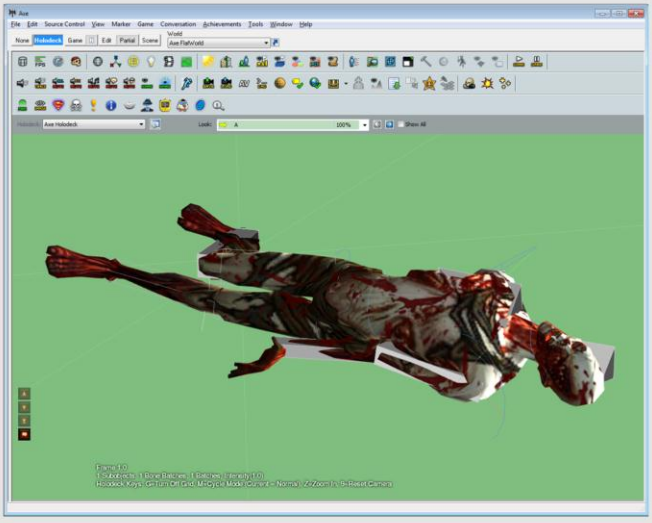
Here's how I build the bone to body map. The map is an array of body indices, one per bone.

Step One: I initialize the map to be all leaf bones.

Step Two: Then I walk through the ragdoll body array and install the body index into the map.

Step Three: For each body, I walk up the tree marking intermediate bones until I hit an ancestor bone with a body.

Ragdolls are tested and tuned in the *Axe* game editor



<explanation>

Let's move on to the game editor. Our game editor, called Axe, has a model viewer called the *Holodeck*. In there ragdolls can be tested and tuned.

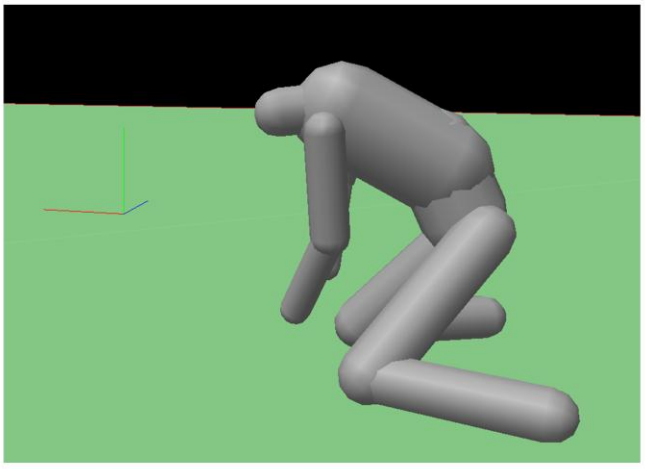
We can view the ragdoll collision shapes and joints to make sure they are working as intended. We can also play back animations that trigger the ragdolls.

Here you see a dead zombie and we have enabled rendering of the ragdoll collision shapes.

Video: zombie ragdoll

Video: zombie ragdoll with collision shape rendering

The model viewer has a mouse joint

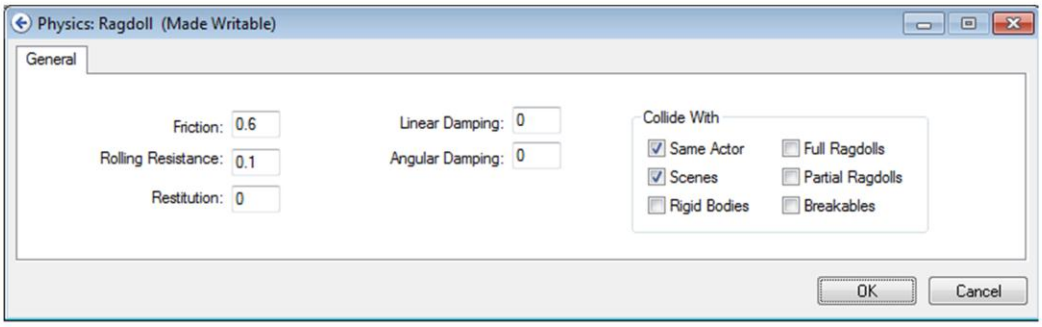


Our model viewer has a mouse joint that lets you move a ragdoll around. This is a simple feature that is quite helpful for quickly testing a ragdoll.

For example, joint limits may need to be tuned to avoid squatting. Dead guys should not be sitting on their knees! The mouse joint can be used to detect bad configurations.

Video: mouse joint

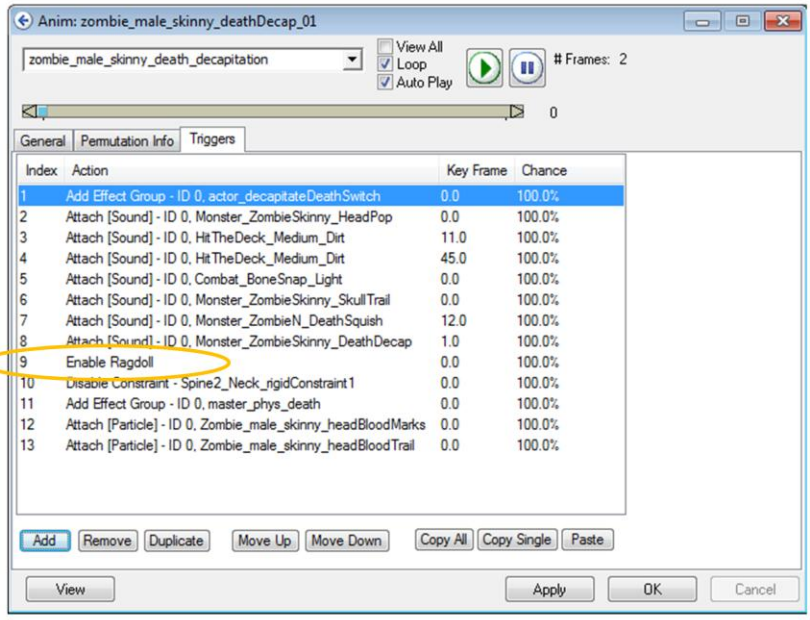
Parameters and collision flags can be tuned



We have some physics parameters that can be tuned in Axe, such as the global friction for the ragdoll. We also can control collision flags in Axe. For example, we decided to improve the performance of the some ragdolls in Diablo 3 by preventing those ragdolls from colliding with each other. However, we can still allow a ragdoll to collide with itself so that a ragdoll by itself looks pretty good.

These parameters live apart from the intermediate file containing the ragdoll geometry. So these parameters can be shared by multiple ragdolls. This makes it easy to do global tuning of ragdoll behavior.

Animation triggers create ragdoll events

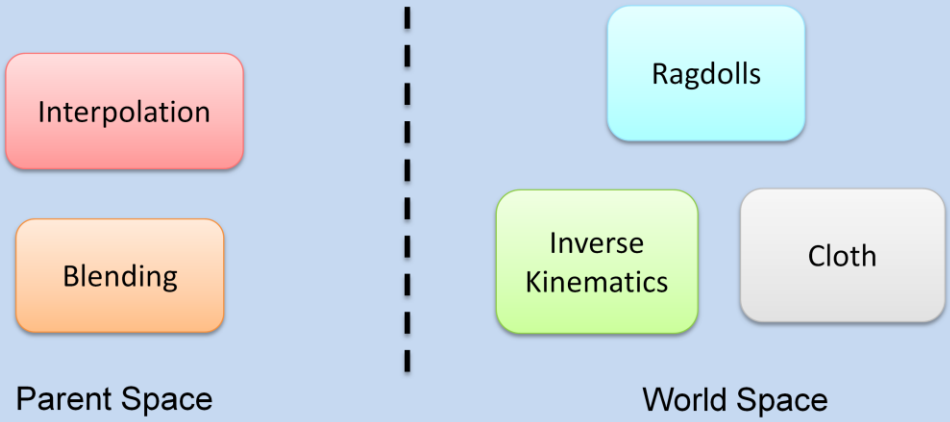


We control ragdoll spawning using animation triggers. This is basically a simple scripting system on a time-line.

Besides spawning ragdolls, we can also disable physics joints and spawn explosions.

You can see more about this system at Julian Love's presentation on Wednesday at 5pm.

Ragdolls are a form of procedural animation



<key point>

So now let's talk about the animation system. When I was integrating ragdolls into the Diablo animation system, it helped me to keep the perspective that ragdolls are just a form of procedural animation.

In our animation system we start with the raw key frames created by the animator. Then we perform interpolation and blending in parent space. Then we compute the bone transforms in model space as I showed before. At this point we apply procedural animations such as ragdoll, inverse kinematics, and cloth.

Procedural animations usually happen in world space because they usually interact with the world in some manner. For example, ragdolls collide with the ground.

We use the ragdoll bodies to adjust the pose

model space keyed animation

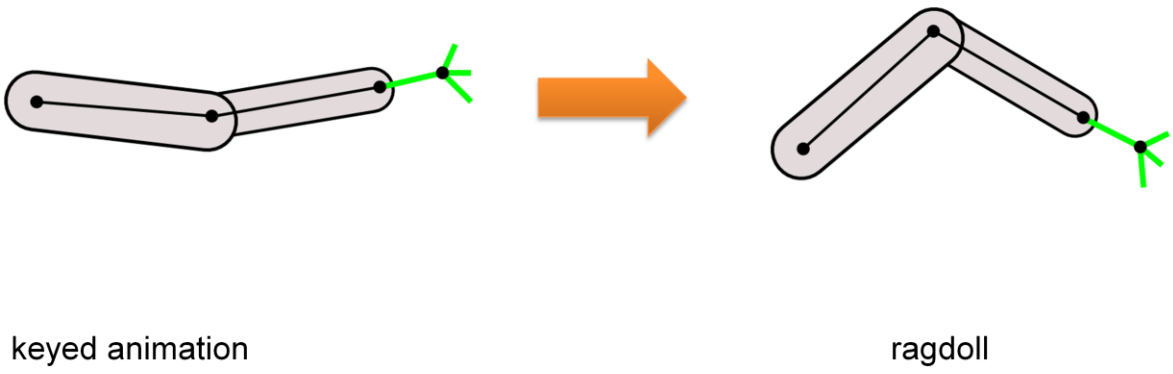
world space ragdoll

<explanation>

As mentioned before, the ragdoll bodies map to a subset of the bones. Bones with bodies follow the rigid body transform exactly. How shall we handle leaf and intermediate bones?

Keep in mind that each rigid body can only drive a single bone. So leaf and intermediate bones need another source of data to fully determine their transforms.

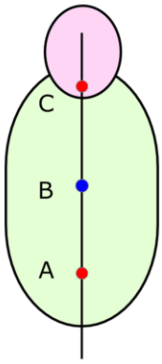
Leaf bones retain their pose relative to their parent



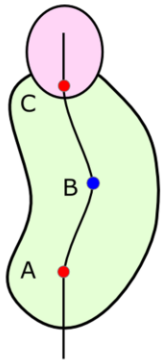
Leaf bones are fairly obvious: they just go along for the ride, holding their relative pose from the keyed animation.

Intermediate bones go to the bind pose

B in bind pose



B in animated pose
→ stretching



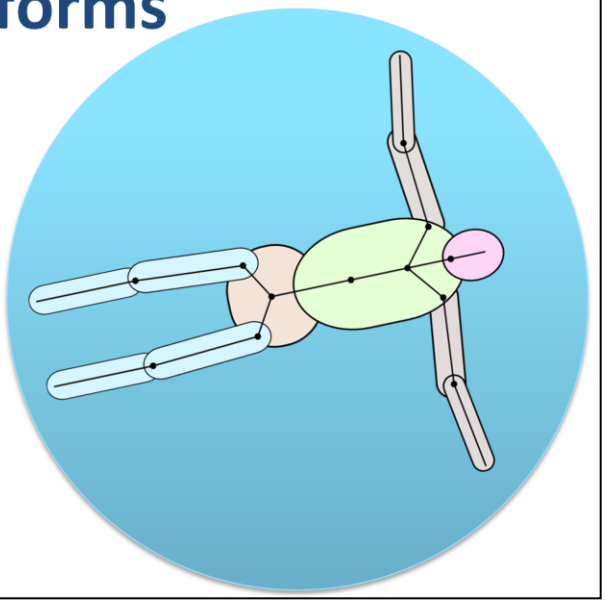
|A-C| = constant

Intermediate bones are a bit trickier. We would like to allow them to be animated, but that doesn't jive with how rigid bodies are connected to each other. Since active bones follow their rigid bodies exactly, an intermediate bone cannot affect its active descendants. Instead you will see the mesh stretch in an unnatural way.

Therefore, we leave intermediate bones in the bind pose relative to their parent. Usually this is not a big problem because we choose bones to be intermediate because they already have limited movement. Nevertheless, your animators should be aware of this when they are constructing animations that have a transition to ragdoll.

Update the actor bounding sphere using the bone transforms

actor transform



In Diablo we maintain a bounding sphere on our actors. This sphere is used for lighting and visibility culling. When a character goes ragdoll they may move very far from the actor transform. They may even fall off a cliff.

Further, we support breakable ragdolls and gibbing of ragdolls, so the ragdoll pieces may be widely separated. So we have to adjust the bounding sphere to contain all the bones, not matter where they are. To do this, we have a local bounding sphere for each bone and sum up all the spheres according to the final bone positions.

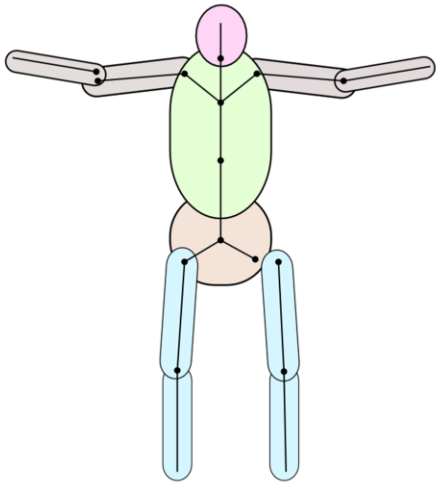
Ragdolls are spawned with care

<explanation>

When a character dies we have to spawn the ragdoll in-place.

We could just create the rigid bodies and joints and cross our fingers. But this can lead to a low quality result.

Spawning in-place causes joints to stretch

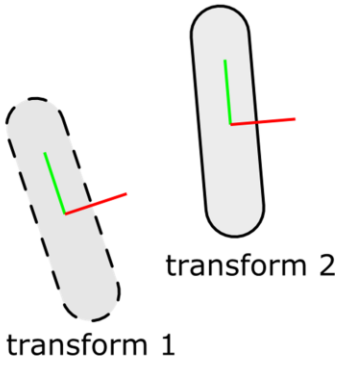


When we spawn a ragdoll, we have to create the bodies where the character's bones are currently located. There is no guarantee that the animated pose conforms to the ragdoll constraints, so joints may be stretched and joint limits may be violated.

We could go through all the joints and try to fix them up by shifting the bodies. However, if you have a good physics engine, it will smoothly eliminate the joint errors over time so you don't have to worry about this problem too much.

But you should keep this issue in mind. If a joint is incredibly distorted, it may not recover. For example, some physics engines may flip a joint 180 degrees if it is distorted by more than 90 degrees.

Obtain the initial velocity from two world space bone poses



$$v = \frac{(p_2 - p_1)}{\Delta t}$$

$$\omega = \frac{2(q_2 - q_1)\tilde{q}_1}{\Delta t}$$

check the polarity!!!

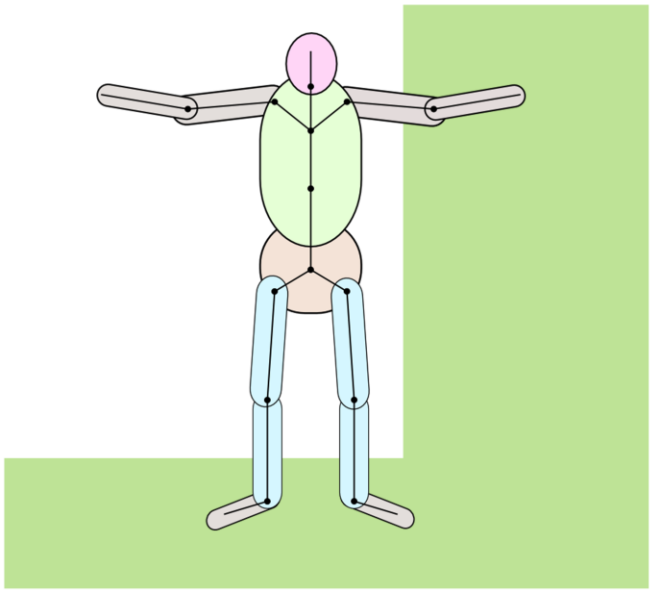
Once the rigid bodies are created we set the initial velocities based on the animation. Otherwise the ragdoll transition will not be smooth. We also allow animators to influence the initial velocity through death animations.

We obtain the initial velocity using two frames of animation. The linear velocity is a straight forward finite difference approximation using two points. The angular velocity is obtained with a finite difference on two quaternions. When you do this, make sure the quaternions have the same polarity.

Video: female zombie spin death

Video: female zombie spin death slow motion

Help, my foot is stuck in the floor!

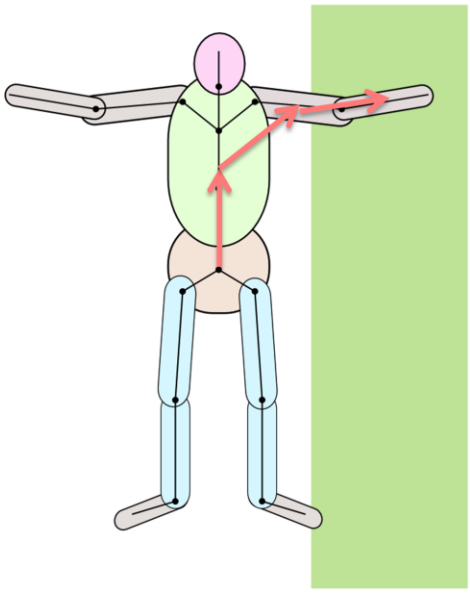


A typical physics bug you see in games is when the ragdoll spawns in an unfortunate location and a limb becomes stuck in the terrain. The terrain is often made from a triangle mesh, so it has no sense of volume. So once a foot is below the ground, it cannot come back out because it bumps into the backside of the triangles. What's worse is the contact constraints start to fight with the joints and this leads to jitter. So your dead guy appears to be convulsing on the ground.

There is no silver bullet for this problem. There are several approaches that can improve the situation. You could eliminate hand and feet collision from your ragdoll. That will solve many cases and I generally recommend eliminating hands and feet because they do not add much visual improvement.

You may also be able to instruct your physics engine to use one-sided collision for the triangle mesh. This will prevent a limb from being stuck and jittering. However, a limb may still stay penetrated in the floor.

Using sequential ray casts to get unstuck



On Diablo 3 I added a routine that tries to remove the initial overlap by translating the entire ragdoll, just enough so that all the center points of each ragdoll collision shape are inside the world. The routine assumes the pelvis is inside the world and then fires rays along the bones, looking for terrain collisions. If it finds a terrain collision, then it shifts the ragdoll to remove the overlap. This proceeds until all the overlaps are removed or an iteration limit is hit.

Tune your ragdolls to improve quality

<key point>

Now, let's talk about tuning. If you want to support lots of different kinds of ragdolls, you will need to work on tuning. Not all ragdolls look good when they are simulated by the physics engine. There are several techniques I'll show you to deal with stability. I'll also show you some ways to get more out of ragdolls then just a simple death.

This little guy is called a *pit flyer*. The pit flyer was one of the most difficult ragdolls to get right in Diablo 3.

I modified our physics engine to enable dumping out ragdolls into a text file that could be loaded into a testbed. There I could look at the ragdoll in detail and try to figure out how to make it behave better.

Video: pit flyer

Video: pit flyer ragdoll

Domino demos

- Pit flyer – unstable
- Pit flyer – fat collision
- Pit flyer – auto-stabilized

Small capsules connecting the wings to the torso make it difficult to simulate the flyer and it has trouble going to sleep. We could make the wing boxes lighter, which effectively makes the arms heavier. But there are two problems with this:

1. Mass tweaking can be difficult for a rigger to get right, so we use a uniform density for shapes based on material.
2. Rotational inertia scales faster with size than with mass, so we can get a better effect by using fatter shapes.

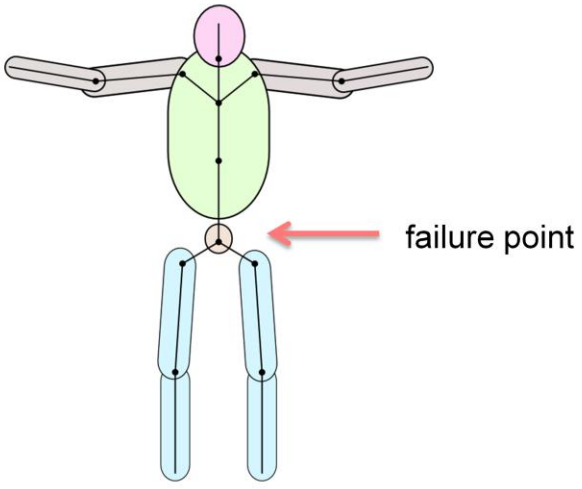
It is easier to spot ragdoll tuning issues visually than by checking the mass.

Mass distribution is crucial for stability

<explanation>

Mass distribution is critical for good ragdoll stability. These rules may not seem obvious at first, but the main point is to design the ragdolls as you would a sturdy structure, such as a bridge.

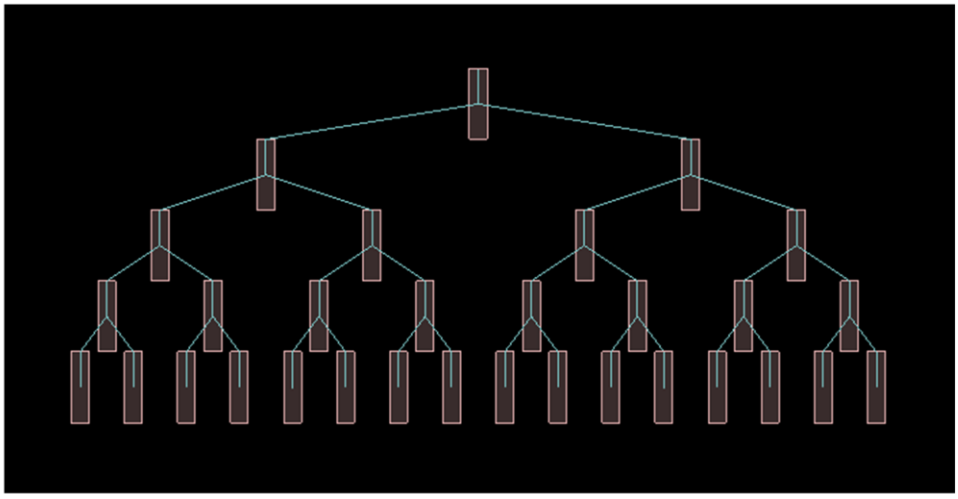
Don’t support bodies with light bodies



Would you build a bridge that is two huge parts connected by a tiny bolt in the middle? Probably not. The same design philosophy applies to ragdolls.

We don’t want to support heavy bodies with light bodies. The opposite is okay. This is due to the nature of the iterative solvers used by most physics engines. They simply cannot converge large forces on small bodies.

Don’t create gaps between bodies



Another design aspect that can hurt ragdoll stability are joint gaps. Gaps are bad for a couple reasons:

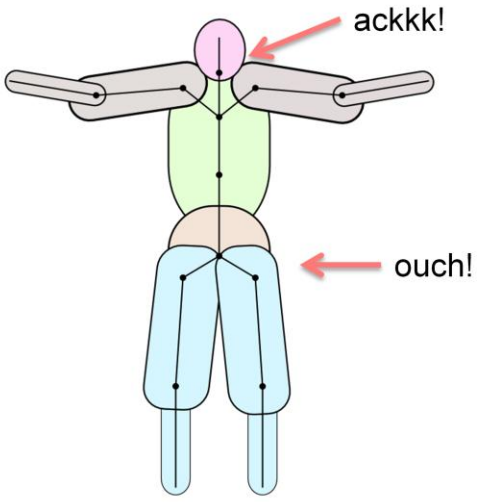
1. other objects can get stuck in a gap, creating a bad visual quality
2. gaps also imply low rotational inertia relative to the torques imposed by the joints. This behaves like a large force on a small body.

Fortunately it is often quite easy for the rigger to close those gaps. In many cases significant overlap works well. For example upper and lower arm collision shapes can overlap around the elbow.

Box2D demo

- Mobile – unstable with gaps
- Mobile – stable with gaps closed

Don’t overlap siblings



In Domino and Box2D I have a joint flag to disable collision between attached bodies. However, bodies that are not directly connected can still collide.

In particular, overlapping siblings can still collide. For example, large thigh capsules can collide. These internal collisions fight with the joints and can make the physics solver unhappy. In Domino, these cause the ragdoll to jitter and propel across the ground.

Domino demos

- Big thighs
- After a diet

Avoid long skinny bodies

Long skinny bodies can be another source of trouble. They have inherently poor mass distribution which can cause them to spin excessively. And they are more difficult for the collision system to handle correctly.

Video: fallen with skinny staff

Video: fallen with fat staff

Ragdoll effects add variety

<explanation>

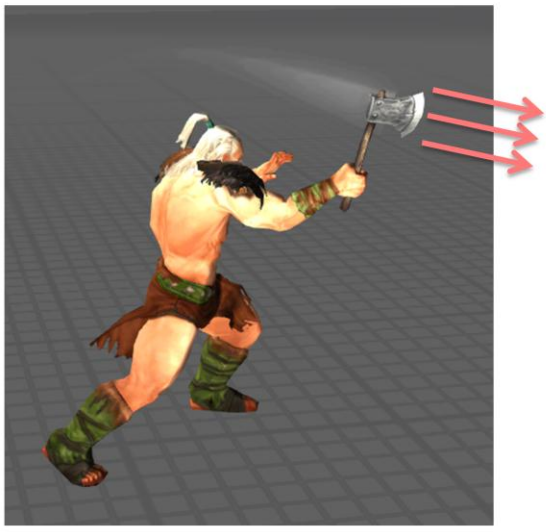
Now I'd like to tell you about some fun effects that we implemented in Diablo 3.

First, we implemented an explosion system to knock ragdolls around.

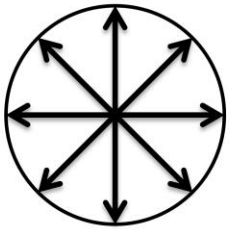
Second, we implemented a system for gibbing those nasty zombies.

Finally, we implemented a system for adding physics simulation to things like pony tails.

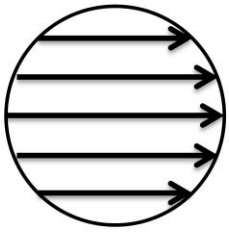
Explosions are physically motivated



radial



linear

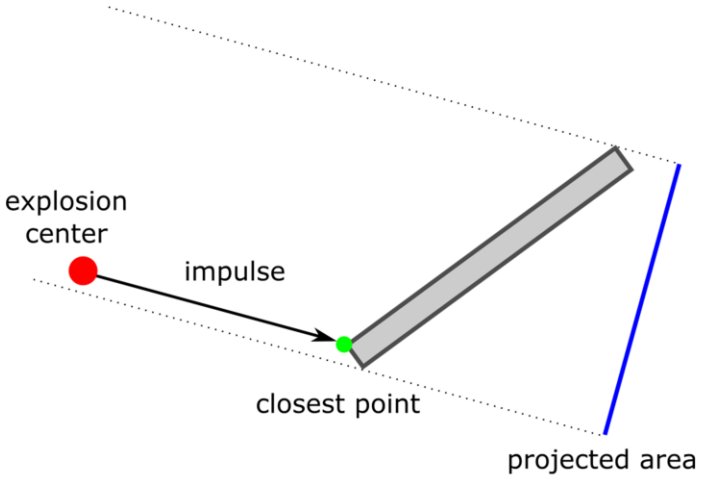


Explosions are crucial for making the world of Diablo 3 feel more interactive. In Diablo 3, stuff is always getting knocked around and blown up.

We have two types of explosions: radial and linear. A radial explosion is the standard spherical explosion that radiates from the center of a sphere. The linear explosion still uses a sphere of influence, but the impulse direction is fixed. The direction can either be in world space, such as *up*, or in local space, such as along the blade of an axe.

Many explosions are set up quite precisely. For example, the weapon swing animation on the Barbarian has a specific contact frame where damage is triggered. At the same time we fire a linear explosion pointed in the direction of the blade. This technique helps to improve the connection between the attack and the ragdoll response.

Explosions account for distance and facing



A ragdoll receives an explosion according to its collision shapes. For each shape we compute the closest point on the shape to the center of the explosion. We apply the explosion at that closest point. This gives the explosion some torque and leads to a nice spinning effect.

Also, we project the surface area of the shape in the direction of the explosion. Then we scale the explosion impulse by the projected surface area. This means that a door facing an explosion gets a bigger impulse than a door pointing away from an explosion.

Video: small explosion

Video: big explosion

Video: zombie attack with explosion rendering

Video: lightning explosions

Video: lightning explosions, debug draw

Gibbing never gets old

Setting up gibbing was one of my favorite tasks on Diablo 3. It works in a fairly direct manner. Death animations are selected from a pool of possibilities. When the dismemberment animation is selected the full ragdoll is spawned, the visual mesh is swapped to a custom gibbed mesh, and then select ragdoll joints are disabled. So we don't have to author separate ragdolls for gibbing.

Video: decapitation

Video: decapitation slow motion

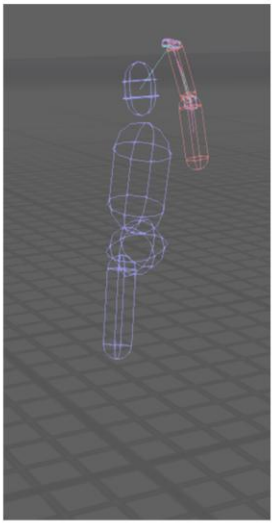
Video: split

Video: dismember

Video: dismember slow motion

Partial ragdolls add flavor to living characters





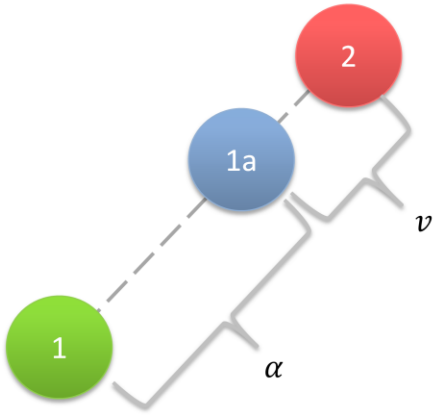
We made a big effort to make the characters in Diablo 3 visually interesting. So the question naturally arose how we could use the ragdoll system to enhance our living characters. Out of this we developed our so-called *partial ragdoll* system.

Our canonical example of the partial ragdoll system is the wizard's pony tail. Here you can see how the partial ragdoll is set up. We have a couple red capsules for the dynamic pony tail. The blue capsules are *kinematic bodies*. Kinematic bodies follow the animation rigidly and provide a collision barrier so that the pony tail doesn't swing through her body.

Partial ragdolls are stabilized with partial teleportation

partial transform: $x_{1a} = x_1 + \alpha(x_2 - x_1)$

velocity command: $v = \frac{x_2 - x_{1a}}{\Delta t}$



The main challenge for simulating partial ragdolls is that characters can move very fast. They can rotate 180 degrees instantly and they can teleport across the screen. However, a physics engine likes smooth movement. These rapid movements can cause the joints to stretch and rip the visual mesh apart. I will briefly explain to you how we dealt with rapid movement.

First of all, every partial ragdoll has a kinematic root body. Kinematic bodies are moved by setting their velocity according to the animation. This leaves the partial ragdoll a frame behind, but we can simply transform the ragdoll bones forward to mask the latency.

We need the kinematic bodies to have a realistic velocity to impart a realistic reaction in the partial ragdoll. However, we need to limit the magnitude of the velocity otherwise joints will rip apart. But then the partial ragdoll with fall behind!

We solve problem this through partial teleportation. We scale down the velocity by teleporting the kinematic body partway forward to the animation pose according to a tuning parameter (shown here as alpha). Then we generate a velocity command to drive the kinematic body the rest of the way

to the animation frame.

Then we apply that partial transform to the entire dynamic sub-tree. This has the effect of subduing the rapid movement while retaining a natural looking reaction. No damping is required!

Videos: Witch Doctor

Video: witch doctor

Video: mojo

box2d.org

@erin_catto

You can download this presentation at box2d.org and you can reach me on twitter or on the Box2D forums.

Thank you for attending my presentation and I hope you find it useful.

I would now like to answer any questions you have.

>>>>>>>>>>>>>>> PARROT THE QUESTIONS <<<<<<<<<<<<<<<<