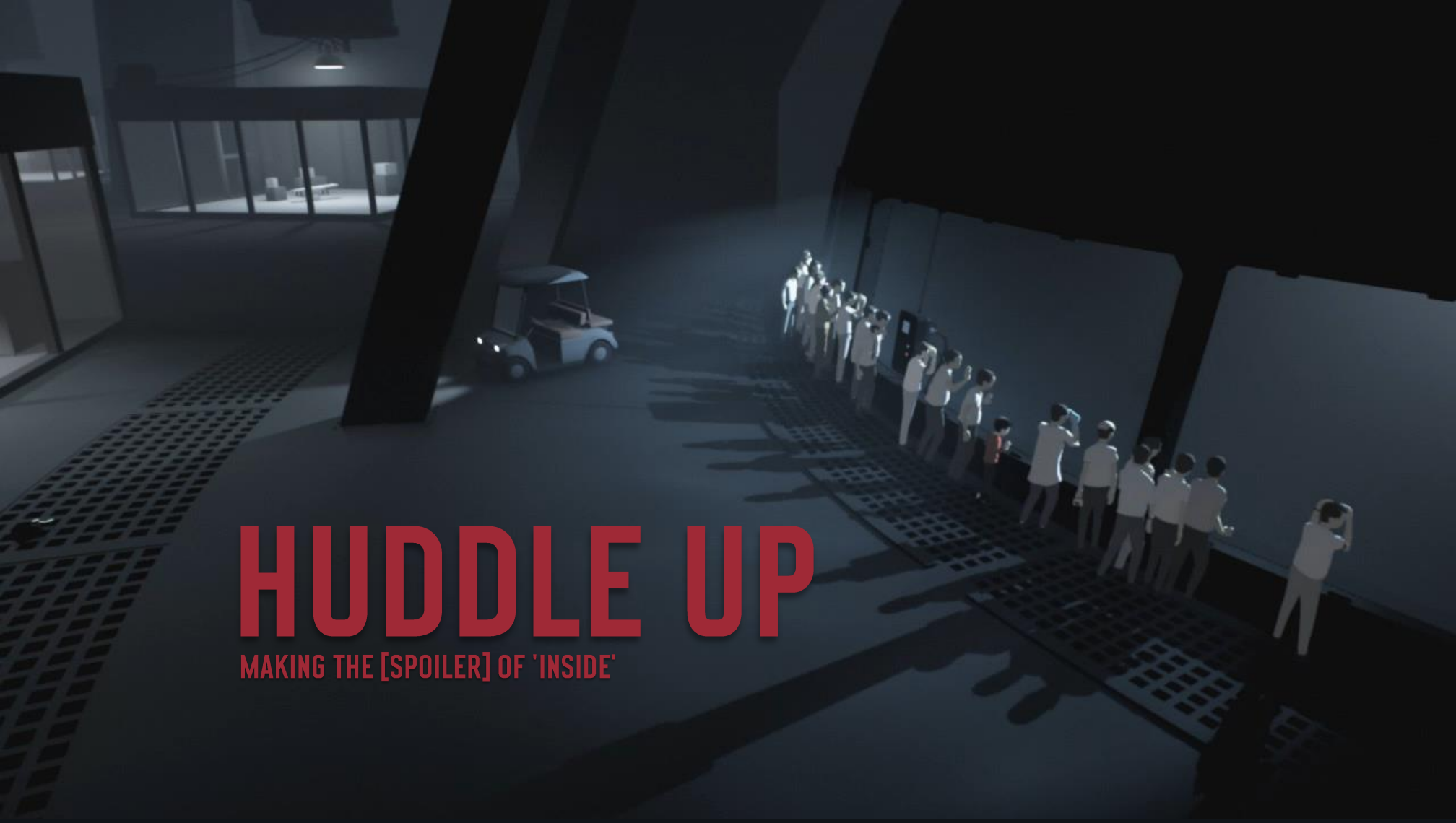


HUDDLE UP

MAKING THE [SPOILER] OF 'INSIDE'

[SPOILER WARNING]

Consider this your final spoiler warning. This talk will destroy your experience with INSIDE if you have not already played it.

Towards the end of INSIDE this happens.
The Huddle – a compound polyhumanoid blob of muscle, fat, skin and bones.
A project-spanning experiment involving constant iteration across several fields.
In the end almost all of Playdead have layed hands on it in some way or another.
We’re going to give you a peek into what makes this monstrosity;

- Movement conceptualization
- Physicallity of the main body or the core
- Handling auxiliary attachements such as arms, legs and torsos
- Shading the beast

PRELIMINARY CONCEPT ANIMATIONS

ANDREAS NORMAND GRØNTVED

My name is Andreas Normand Grøntved and I am the animator at Playdead.
In 2011 I was hired to work with Playdead to do the animations for INSIDE.
But before that, in 2010, I was contracted to do preliminary concept animations for the Huddle.

”THE” DRAWING OF THE HUDDLE

This is one of the first drawings of the Huddle. It is drawn by our lead artist Morten Bramsen.
This drawing is often referred to as “the” Huddle drawing because throughout production we would refer to it when making decisions about shading, lighting, modelling and overall style – not limited to the Huddle, but in general.
While it is visually distinctive it does not offer much detail on motion other than what is naturally embedded in an image.
My task was to do preliminary motion tests for it.
As an animator the first thing I thought about when seeing this drawing was ->

INSPIRATION

The demonized boar god Nago that chases Ashitaka through the woods in the beginning of Princess Mononoke.
It scrambles aggressively towards its target. Violent, determined.
Most importantly it morphs and recalibrates to accommodate its mission.
If it needs an arm or a leg it'll spawn it to cover that need.

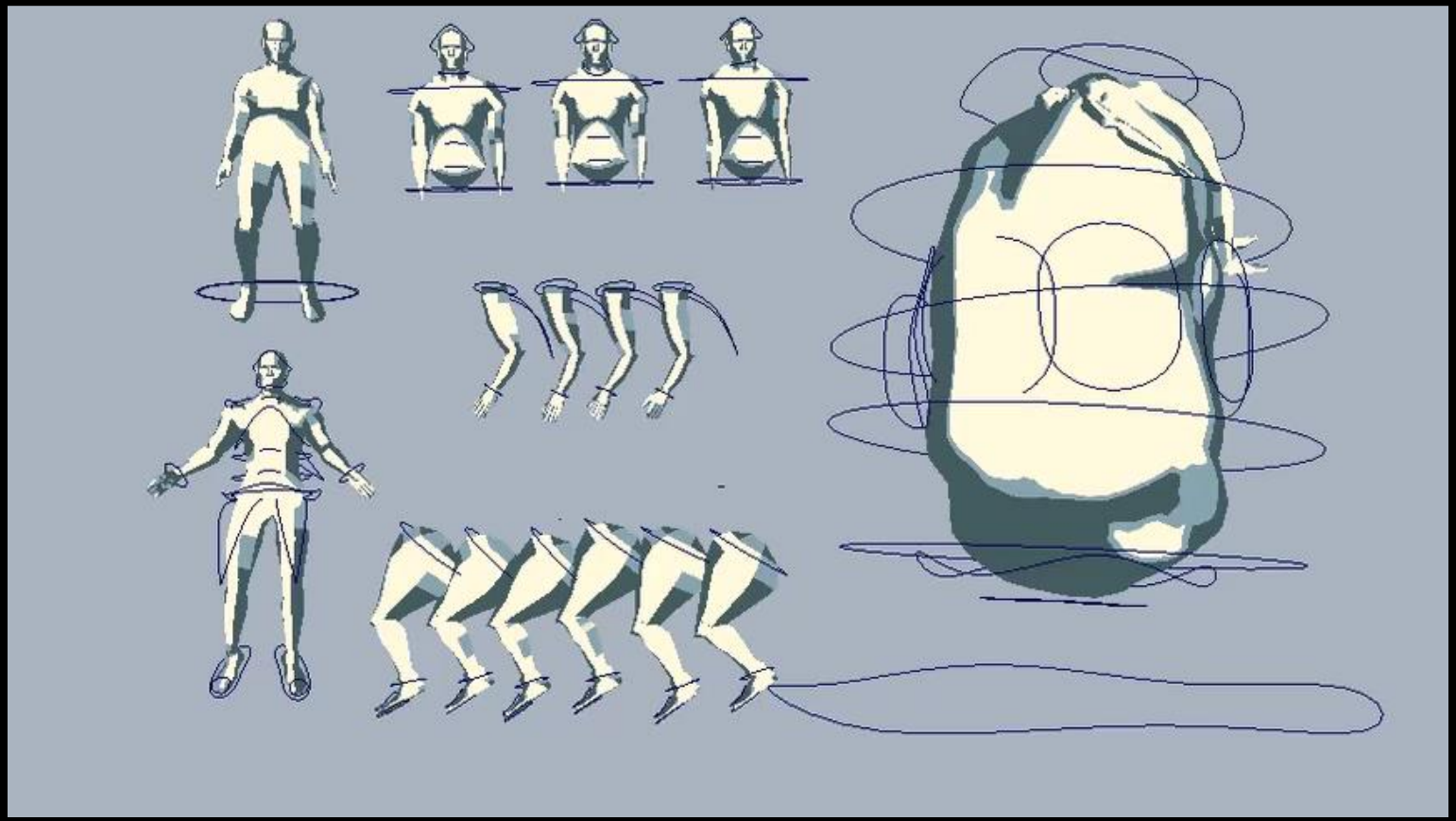
… ALSO THIS

Another thing I couldn’t stop thinking about was the old physics-driven stickiness game Gish.
It has this squishy, sticky and bouncy little oil blob that deforms and flexes to get through the levels.
If one add arms, legs and general disgust to this little fellow, it’ll become a small, charming bouncy Huddle.

..... AND THIS

Finally a great inspiration for how the arms and legs should behave could be found in crowd-surfing.
In a crowd-surfing crowd the many individuals share a common goal, but they act slightly differently from one another.
Some will gently support the surfer, others will try to keep the surfer fixed in one spot and finally some will downright attempt to tear down the surfer.
This mixture of motives really related to how the Huddle works. A group of individuals strung together – a group that shares a common goal; that of the player.
These were the pillars of inspiration for the preliminary movement tests.

HUDDLE POTATO



So I built this test Huddle. It is quite a rough potato rig.
It has 4 spinal bones, about 20 surface bones to drive the skin.
4 arms, 6 legs, 3 torsos and 2 full bodies. All are free-floating which means I could retract them into the potato and sprout them somewhere else on the body if needed.
It is quick to animate do to the low complexity.

So here comes a bunch of animations done to sort of see how the Huddle could move and behave.
Please disregard the visuals, lighting, shading and overall quality of the videos. This is about movement.

FIRST TIME STANDING UP

This is the first test from late 2010.

FIRST STEPS

Here are the Huddles first steps.
Martin Stig Andersen was kind enough to add some sound from his elaborate recordings of oily naked people getting slabbed by large pieces of meat.
Sadly this is the only animation that received that treatment.

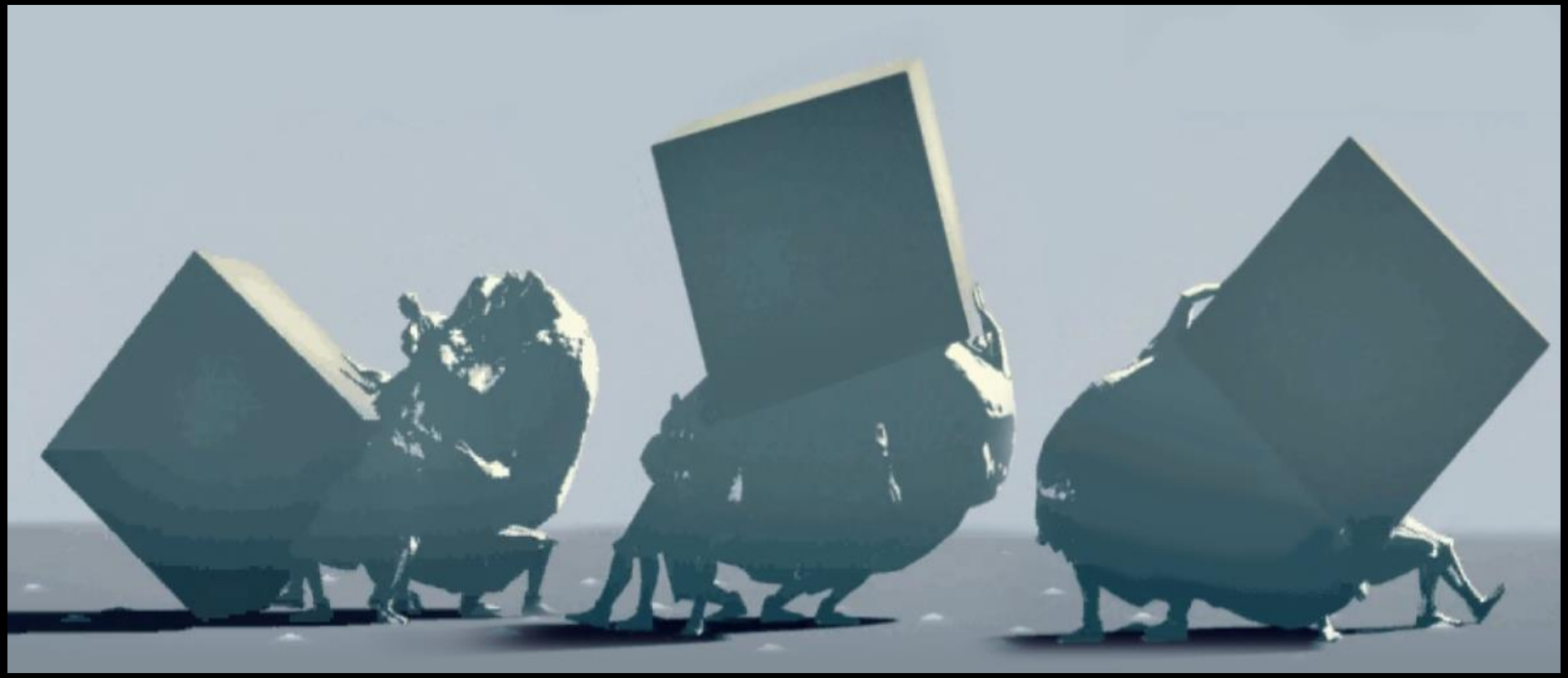
PEEING

My plan was for the Huddle to have several different idle activities for when there was no input from the audience.
Peeing could be one of those activities.

IDLES

Here’s a potpourri of simple idle animations.
The idea of bird landing on the top part of the Huddle is preserved throughout production and is included in the end of the game, where several birds can land on the Huddle.

LIFTING LARGE CRATE

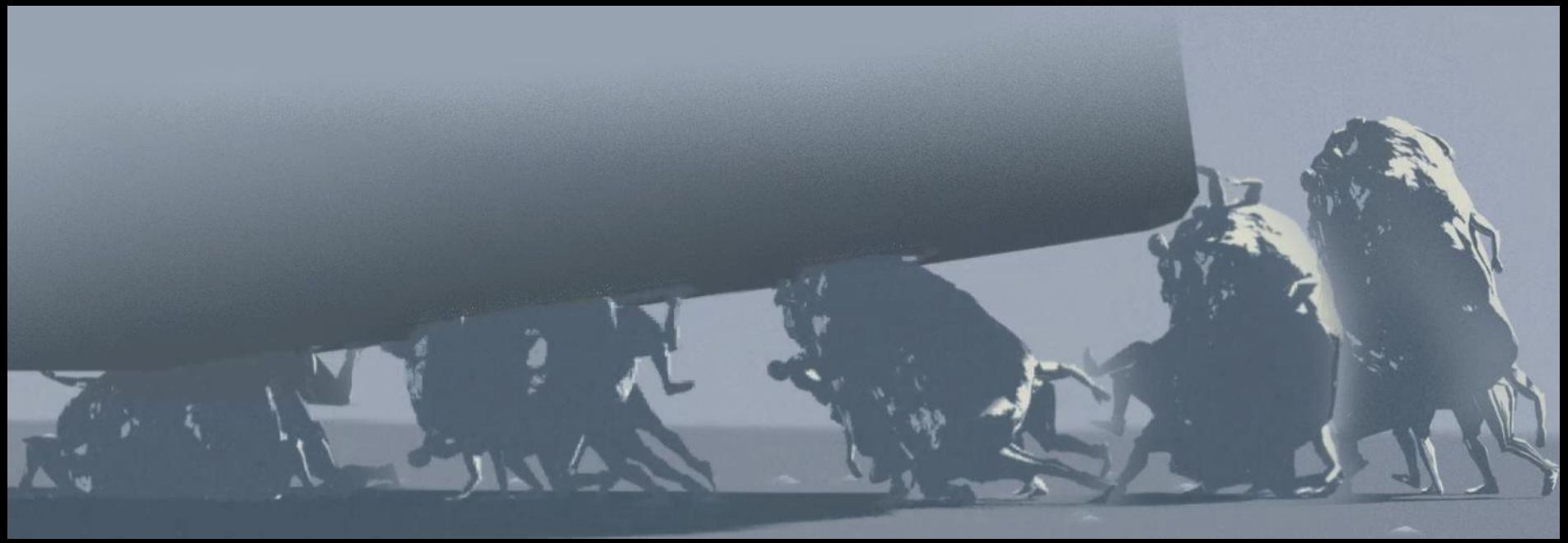


The Huddle lifting a large crate. This is reproduced in INSIDE as well. The spot where the Huddle rips out a server from the wall to use it to gain access to a tall ledge.

REACHING LEDGE

Here the Huddle reaches a tall ledge.
The scrambling behaviour is evident here – see how the legs try to add helpful forces to the body even though they do not have any ground contact or contact with the wall.

THROUGH NARROW SPACES



We would obviously utilize the natural squishynes of the Huddle to get it through narrow gaps in the wall.
Sadly we ended up deleting a scene similar to this one, but we did implement a small door and a vertically aligned hatch for the Huddle to push itself through.

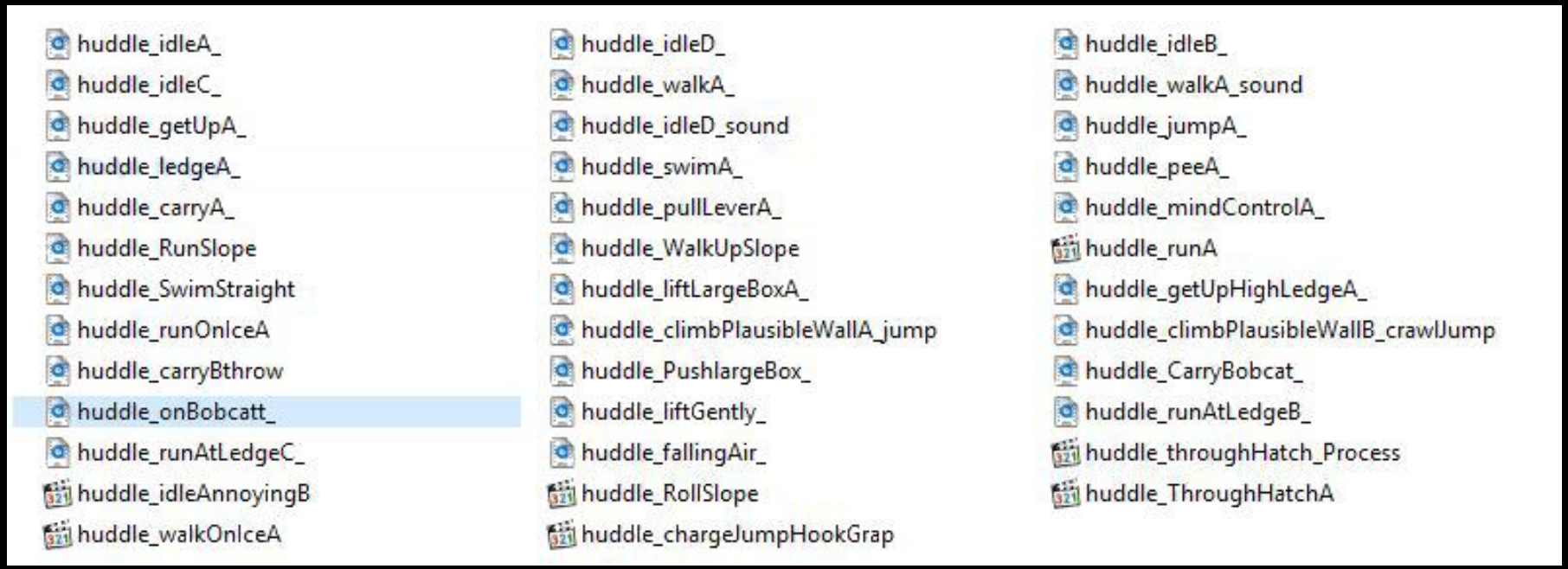
WALKING ON ”ICE”

Here is a test of the Huddle walking on ice.
We never implemented any ice in INSIDE, but we do have a newly mobbed floor in the aforementioned server-scene,
and the spot where the Huddle converts the CEO to a bloody mess it can skit around in the pool of blood that remains.

CHARGE JUMP – GRAB HOOK

One of the last concept animations I did was this early inspiration for the pendulum scene where the Huddle hijacks a swinging crane to swing back and forth to rip out a piece of wall so it can get through it.
This concept animation tests a couple of things such as squash/stretch, a small jump and the silhouette/plausibility ratio.
We would discuss how many arms was needed to carry the Huddle versus how many arms would look nice. Obviously the Huddle would need maybe 20 arms to carry its weight in a situation like this.
However it is important to preserve the silhouette to keep it visually readable in the frame. We didn't settle on any specific number of arms, but we did choose to keep it down.

ALL THE TESTS



Before I started to work on the actual game I did about 40 of these Huddle concept animations.
They gave us a sense of where we were heading. I like to think the animations contributed as inspiration for the team, a spot on the horizon to aim for.
Needles to say the team outdid these concepts by miles.

IF NOT SO. THEN HOW?

The broad audience thinks the Huddle consists of 1000s upon 1000s of animations strung together in sequence.
That is not true not is it feasible. The Huddle isn’t animated in a classical sense, so I didn’t make it.
Next up: Lasse and the physicality of the Huddle body, the core.

Part 2: Huddle core

Lasse Jon Fuglsang Pedersen
Programmer // Playdead

hi, my name is Lasse, and i'm a senior programmer at playdead

I'm going talk a bit about the huddle core physics
--i.e. what makes it tick underneath

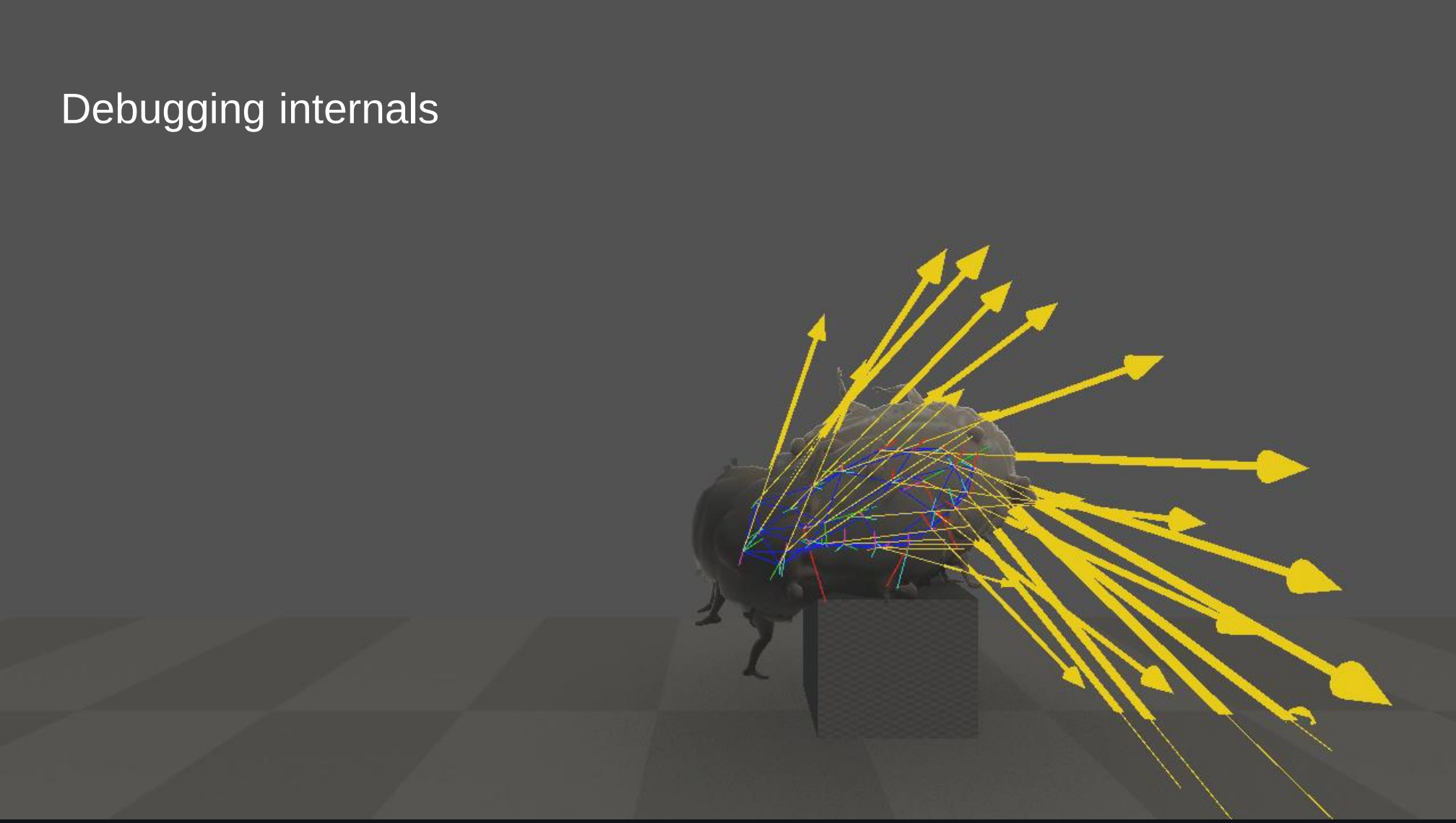
Preface

- The Huddle core is animated through physics simulation
  - Skinned mesh bound to dynamic bodies
- Custom physics model developed by Thomas Krog
  - Read and analyze world state
  - Generate internal impulses
  - Sync to physics engine
- Central in set of largely independent systems
  - Independently generating impulses
  - Ripe for “emergent” behaviour
  - Finely tweaked for feel

as Andreas mentioned, the huddle core isn’t animated by conventional means
--it still has a skinned mesh, but its skinning bones are governed by physics rather than animation curves

I would like to show a short video with some debug overlays of internal impulses and velocities
--just to give you an idea of the amount of acting forces

Debugging internals



[30 second video clip of internals]

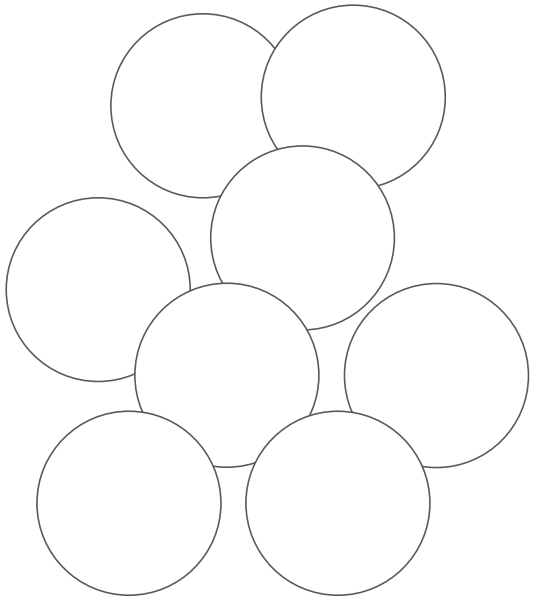
so there’s a lot going on, lots of acting forces

Model overview

let’s go through the different components of the model
--dissect it a bit

Model overview

- N dynamic bodies
  - Sphere colliders
  - Updated by physics

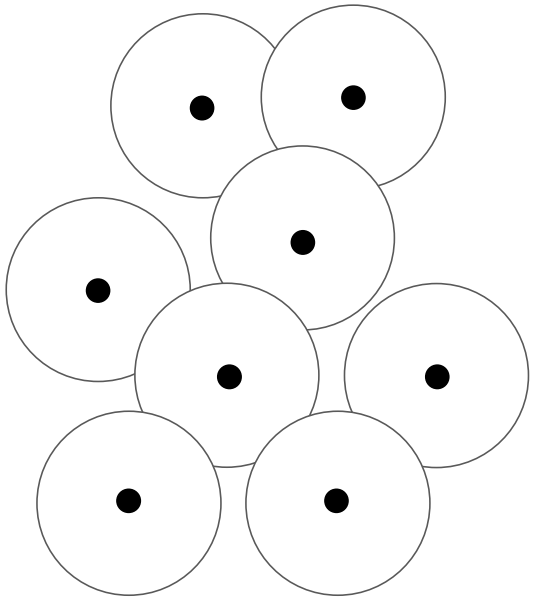


first of all we have a bunch of dynamic bodies
N=26

these live in the physics engine, and each body has a sphere collider attached to it

Model overview

- N dynamic bodies
  - Sphere colliders
  - Updated by physics
- N internal bones
  - Sources dynamic bodies
  - Cache position, velocity, mass, …
  - Accumulate internal impulses

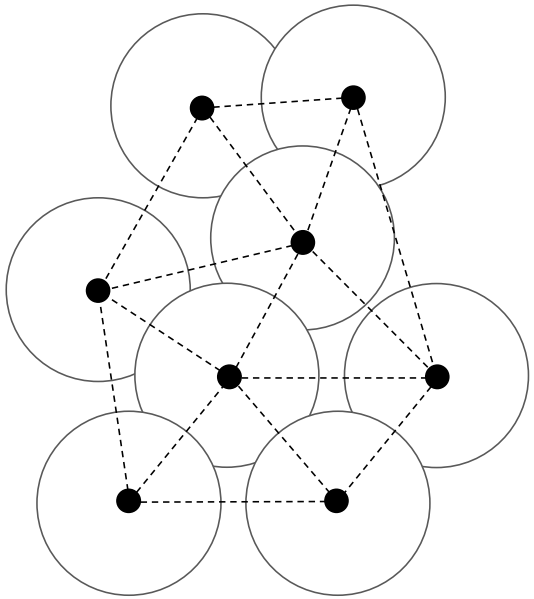


on top of the dynamic bodies, we have 26 internal bones
--each internal bone is bound to a dynamic body

these cache various properties of the dynamic bodies
--also serve to accumulate the internal impulses that add up during the internal time step

Model overview

- N dynamic bodies
  - Sphere colliders
  - Updated by physics
- N internal bones
  - Sources dynamic bodies
  - Cache position, velocity, mass, …
  - Accumulate internal impulses
- Adjacency graph
  - Edges between close neighbours

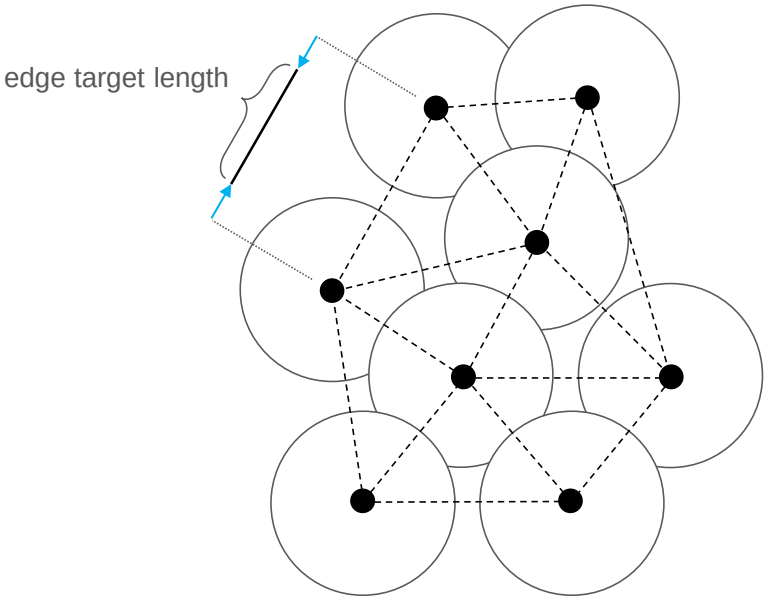


on top of the internal bones we have an adjacency graph, or an internal mesh if you will
--set of edges established between closest neighbouring internal bones

the ”closest neighbouring internal bones” are decided on game start
--at this time, the internal bones are distributed on a sphere, with equal spacing

Model overview

- N dynamic bodies
  - Sphere colliders
  - Updated by physics
- N internal bones
  - Sources dynamic bodies
  - Cache position, velocity, mass, …
  - Accumulate internal impulses
- Adjacency graph
  - Edges between close neighbours
  - Deformable target lengths



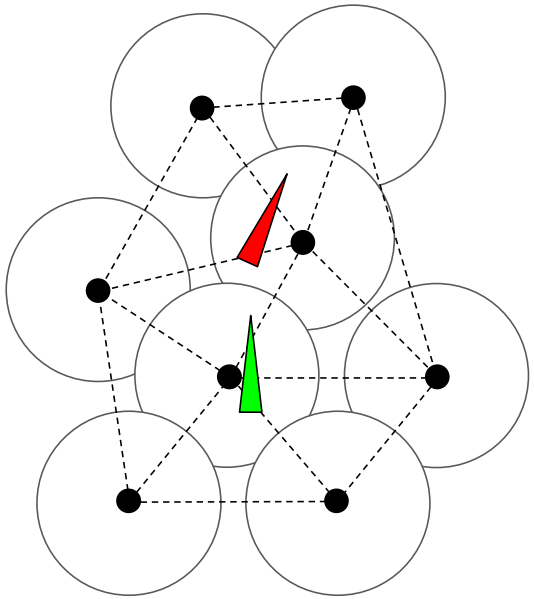
every edge has a target length
--acts as spring between pair of internal bones

target lengths continuously deform, based on scale and total height of the system
--imagine vertically stretching a sphere

spring impulse: r/||r|| * (dt * mass * strength * (||r|| - targetLength))

Model overview

- N dynamic bodies
  - Sphere colliders
  - Updated by physics
- N internal bones
  - Sources dynamic bodies
  - Cache position, velocity, mass, …
  - Accumulate internal impulses
- Adjacency graph
  - Edges between close neighbours
  - Deformable target lengths
- 2 spine bones
  - Driven by logic and animation

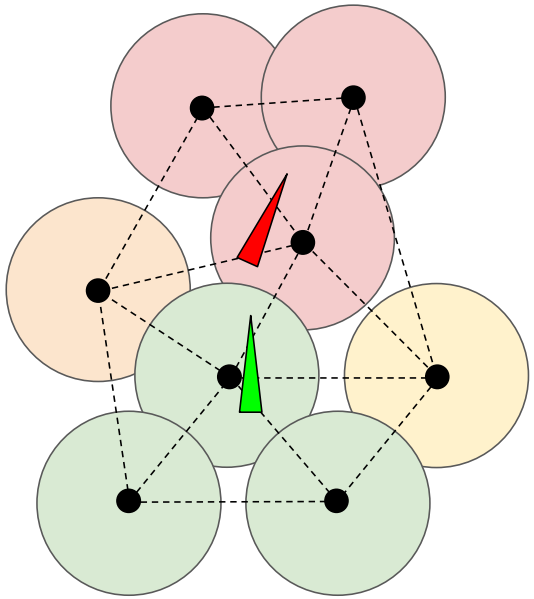


finally, we have a pair of spine bones: top and bottom
--these are neither pure physics or pure animation

they are simulated internally, and they are also affected by animation curves

Model overview

- N dynamic bodies
  - Sphere colliders
  - Updated by physics
- N internal bones
  - Sources dynamic bodies
  - Cache position, velocity, mass, …
  - Accumulate internal impulses
- Adjacency graph
  - Edges between close neighbours
  - Deformable target lengths
- 2 spine bones
  - Driven by logic and animation
  - Each affects a cluster of internal bones



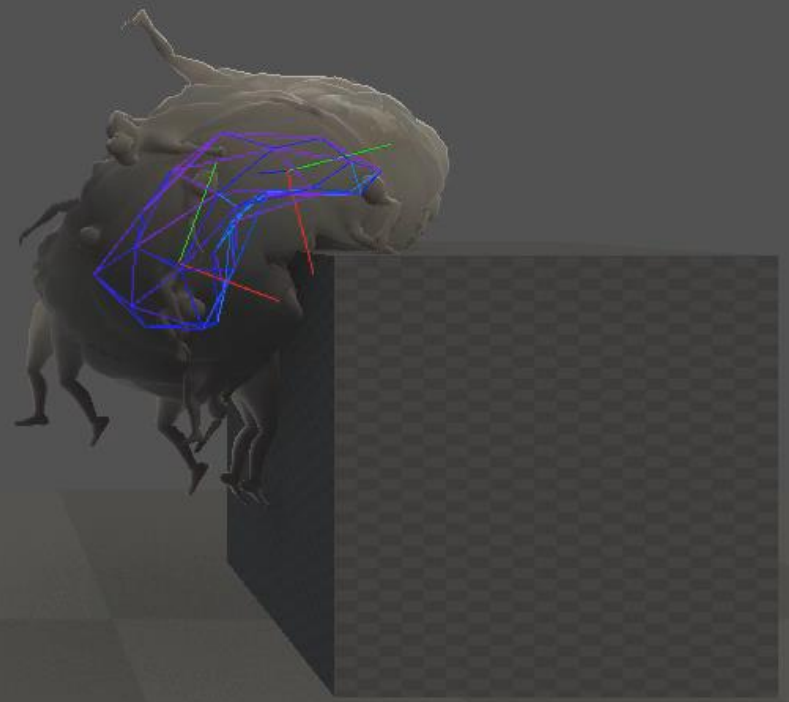
internal bones are given spine bone weights by vertical distance to each spine bone

spine bones act as local origins for associated bones, and generate impulses to (loosely) maintain local pose
--e.g. if top spine bone rotates, will generate pose-matching impulses for entire top of the huddle

also, direct impulses to a spine bones are translated into impulses to all associated bones
--can use spine bones to model ”macro-level” motion

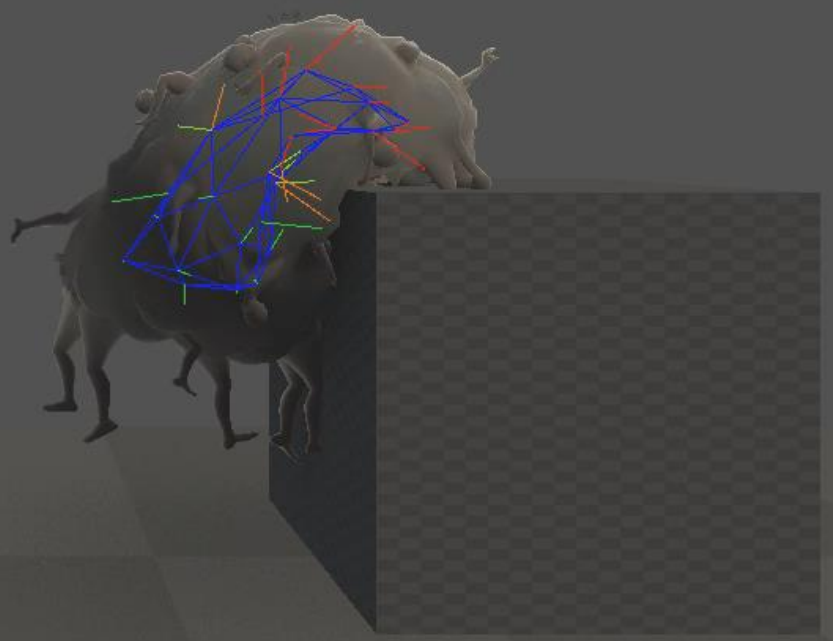
let’s take a look at some debug overlays again

Internals during climb



here we see the spine bones, and the internal mesh
--can see how edges continuously tense up and relax

Internals during climb



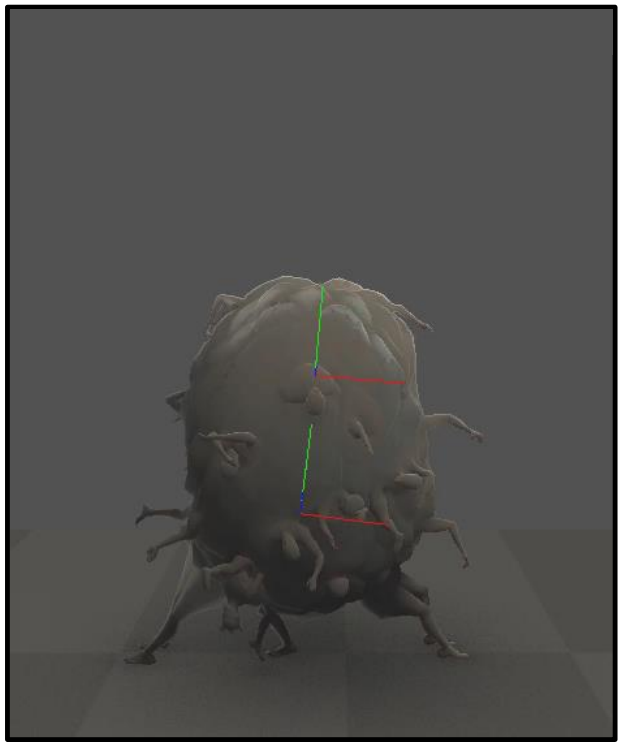
and here we see the impulses from the spine bones
--colored by spine bone

Reconfiguring spine

would like to talk a bit more about the spine, and how we reconfigure the spine at runtime
--essential part of the model

Reconfiguring spine

- Spine is the Huddle’s up-vector



huddle is standing when spine facing up
--has entire system dedicated to maintain its balance

[video showing spine while standing]

Reconfiguring spine

- Spine is the Huddle’s up-vector
- Tends to roll and fall rather clumsily
  - In most landings, spine ends up sideways
  - How do we get up from this pose?

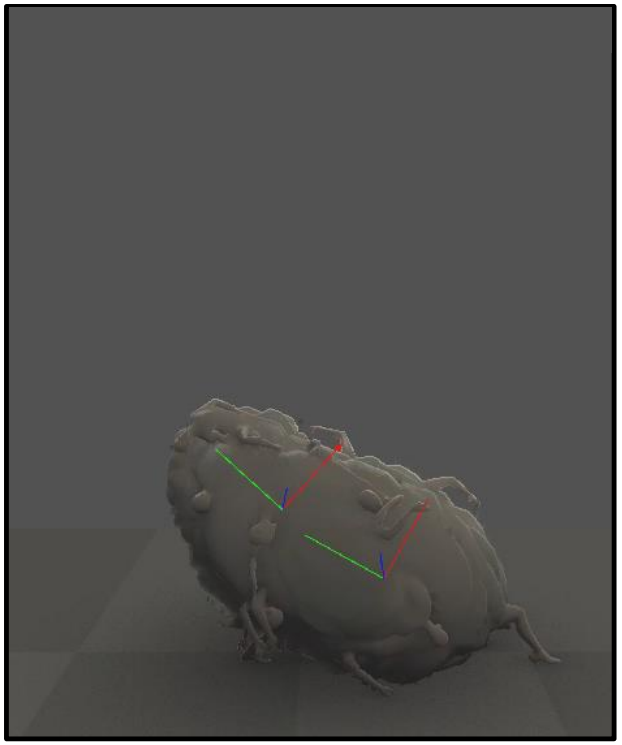
moving around depends on standing pose, i.e. upwards-facing spine
-- but spine often ends up sideways

clumsy by design (”wants” to roll over edges, fall over, etc.)
--need to get up from this pose to keep moving

[video showing spine after landing]

Reconfiguring spine

- Spine is the Huddle’s up-vector
- Tends to roll and fall rather clumsily
  - In most landings, spine ends up sideways
  - How do we get up from this pose?
- Could rotate to stand, but looks a bit silly

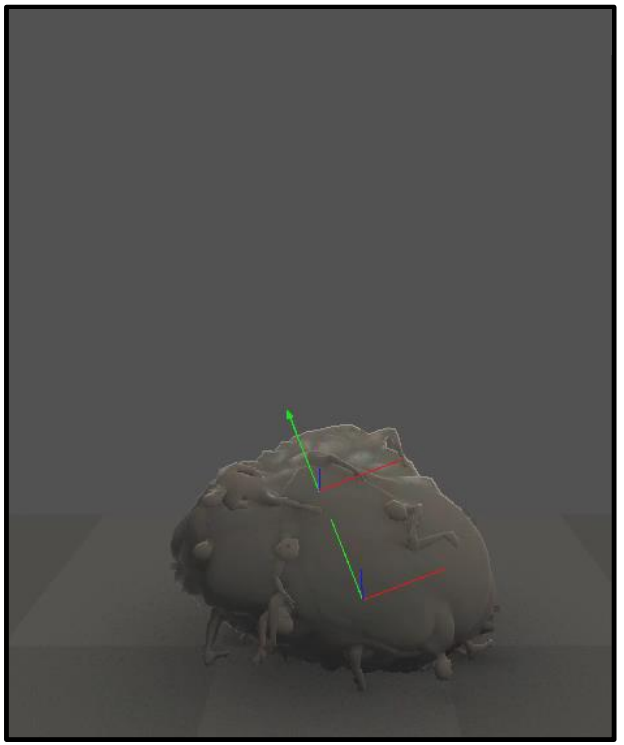


can apply torque to spine bones, rotate to stand
--doesn’t look very good, doesn’t respect ”weight” of system

[video showing “getup” by rotation of spine]

Reconfiguring spine

- Spine is the Huddle’s up-vector
- Tends to roll and fall rather clumsily
  - In most landings, spine ends up sideways
  - How do we get up from this pose?
- Could rotate to stand, but looks a bit silly
- Reconfigure spine bones to rise vertically

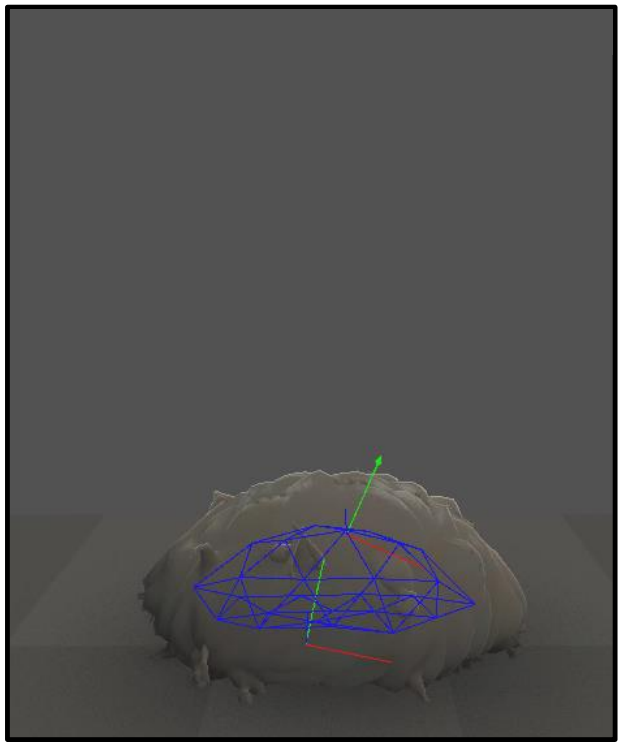


instead we reconfigure spine, and reposition the spine bones vertically
--requires recomputing associated internal bone weights, skinning positions

[video showing “getup” by reconfiguring spine, rising vertically]

Reconfiguring spine

- Spine is the Huddle’s up-vector
- Tends to roll and fall rather clumsily
  - In most landings, spine ends up sideways
  - How do we get up from this pose?
- Could rotate to stand, but looks a bit silly
- Reconfigure spine bones to rise vertically
  - Visible pop in case of sudden deformation



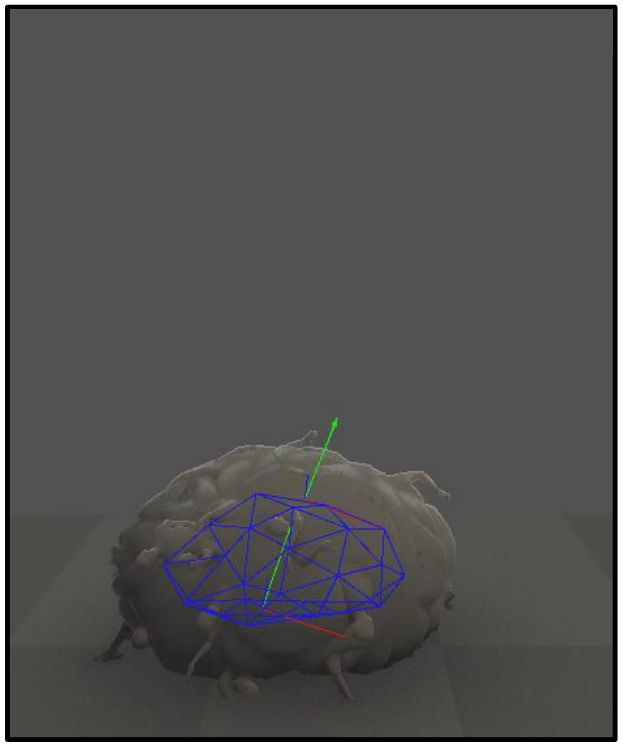
reconfiguring spine bones also changes total height of system, often by quite a bit
--very sudden change in target edge lengths

results in strong discontinuity in the forces acting on the model from frame to frame
--and we get a very visible pop (donotwant!)

[video showing “getup” by reconfiguring spine, rising vertically, with pop]

Reconfiguring spine

- Spine is the Huddle’s up-vector
- Tends to roll and fall rather clumsily
  - In most landings, spine ends up sideways
  - How do we get up from this pose?
- Could rotate to stand, but looks a bit silly
- Reconfigure spine bones to rise vertically
  - Visible pop in case of sudden deformation
- Blending edge lengths to mask the pop



to fix discontinuity, we blend the target edge lengths over a short amount of time
--occurs only when reconfiguring spine bones, but this is actually quite often

[video showing “getup” by reconfiguring spine, rising vertically, with edge length blends]

A network of impulses

... coming back up to surface level ☺

A network of impulses

- The Huddle is comprised of many systems
  - Can be divided into core, logic states, and auxiliary

many different types of systems make up the huddle

A network of impulses

- The Huddle is comprised of many systems
  - Can be divided into core, logic states, and auxiliary
- Core is responsible for holding the model “together”
  - Maintains edge lengths and drives spine bones
  - Receives impulses from other systems
  - Talks to the engine

the huddle core is what keeps it all together
--defines the constraints on the desires of everything else

A network of impulses

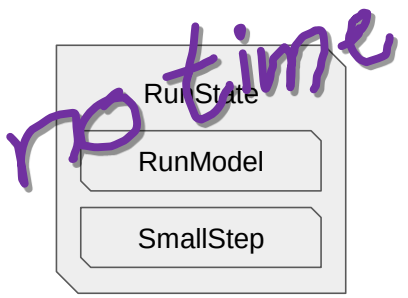
- The Huddle is comprised of many systems
  - Can be divided into core, logic states, and auxiliary
- Core is responsible for holding the model together
  - Maintains edge lengths and drives spine bones
  - Receives impulses from other systems
  - Talks to the engine
- Control impulses come mainly from logic states

what makes the huddle walk, run, climb, swim, etc.?
--this is handled by different logic states

each logic state responsible for generating and applying control impulses for a specific type of behaviour
--have to keep in mind that every impulse is just the desire of one particular system

A network of impulses

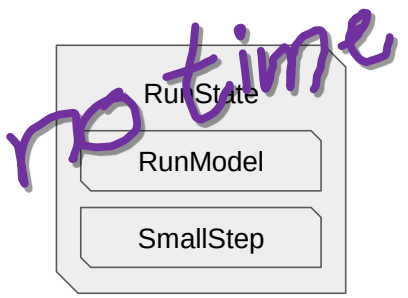
- The Huddle is comprised of many systems
  - Can be divided into core, logic states, and auxiliary
- Core is responsible for holding the model together
  - Maintains edge lengths and drives spine bones
  - Receives impulses from other systems
  - Talks to the engine
- Control impulses come mainly from logic states
  - Some really complex, deserve time on their own



no room to cover individual states here
--many of them really complex, the result of years of experiments and iterations

A network of impulses

- The Huddle is comprised of many systems
  - Can be divided into core, logic states, and auxiliary
- Core is responsible for holding the model together
  - Maintains edge lengths and drives spine bones
  - Receives impulses from other systems
  - Talks to the engine
- Control impulses come mainly from logic states
  - Some really complex, deserve time on their own
- Auxiliary systems
  - Visuals, audio, level-specific, all the lovely details

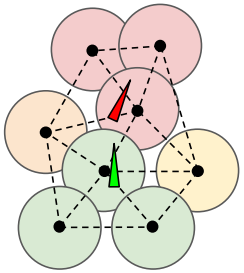


finally, there's a host of auxiliary systems
--some are pure output systems (analyzing behaviour, rather than trying to modify), while others are generating impulses

for example:

game designers suggested having the huddle consistently touch down on the corner of an edge when running off the edge
--there's a system in place for this, which is always looking at the bottom-most bone's trajectory towards the closest approaching edge, and applies impulses towards make it hits the corner

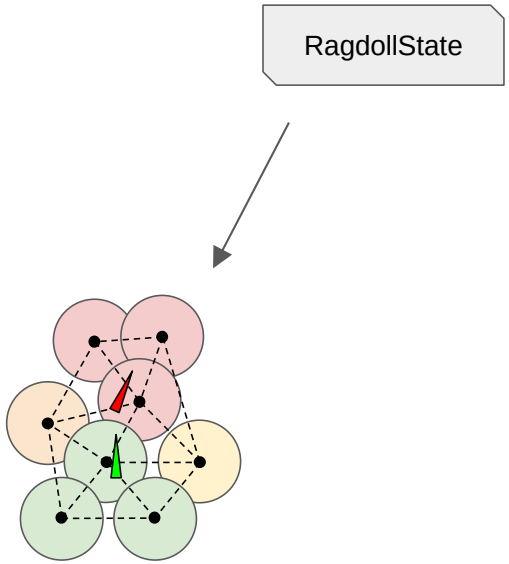
A network of impulses



we can also look at it like this
--essentially, the huddle is a network of impulses

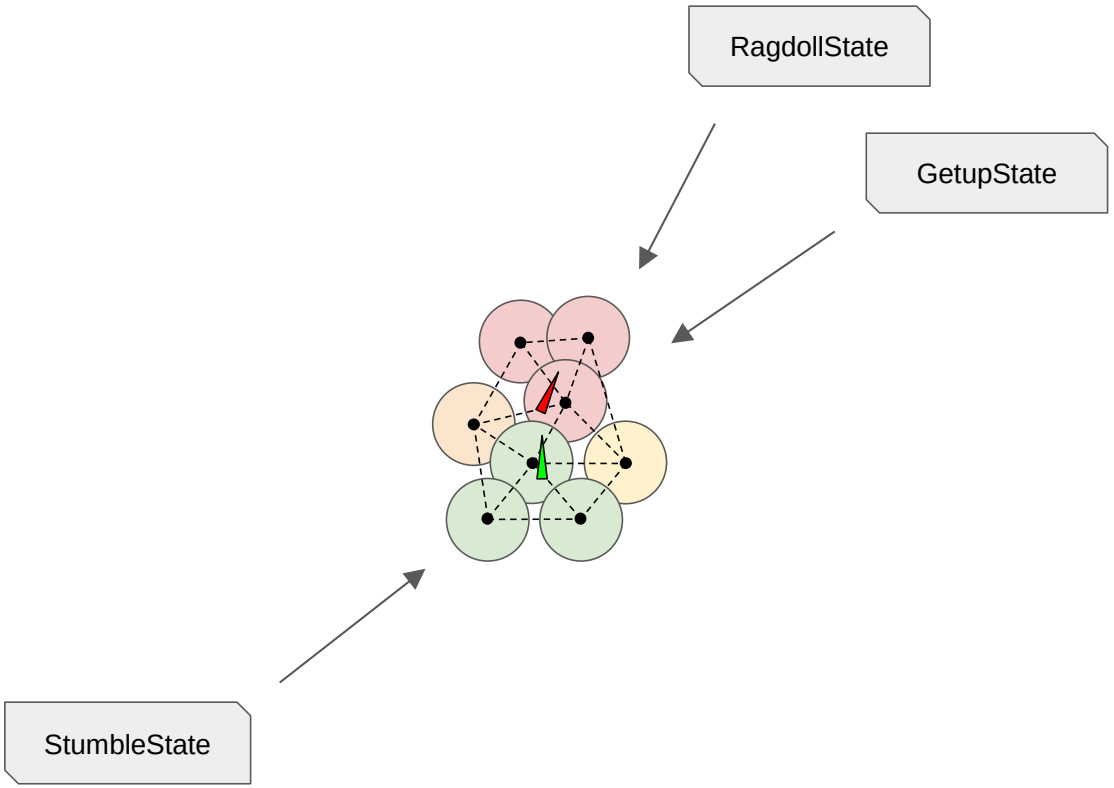
first, here is the core with all of its internal impulses for maintaining edge lengths etc.

A network of impulses



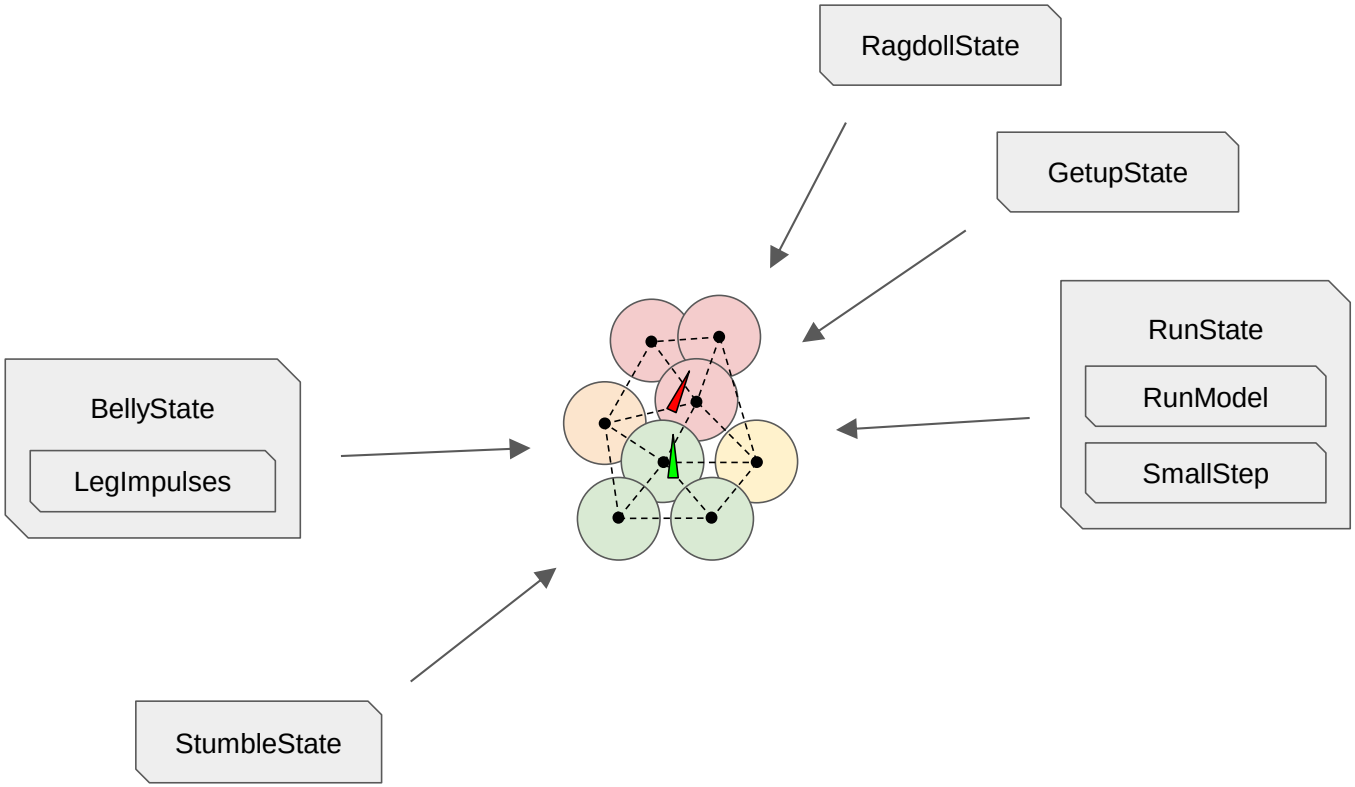
and here we have a logic state
--it applies control impulses to the core

A network of impulses



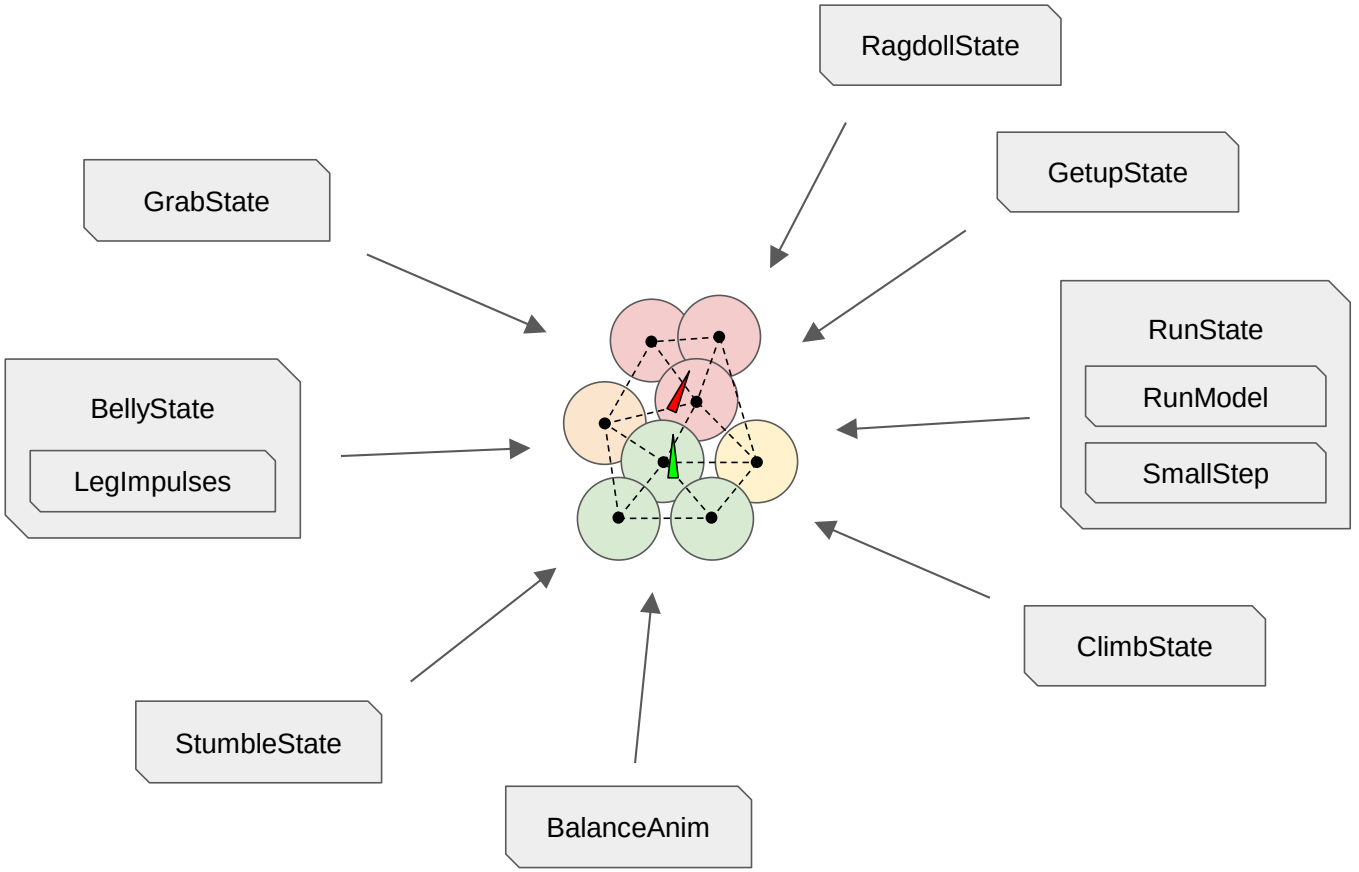
and there are more logic states ...

A network of impulses



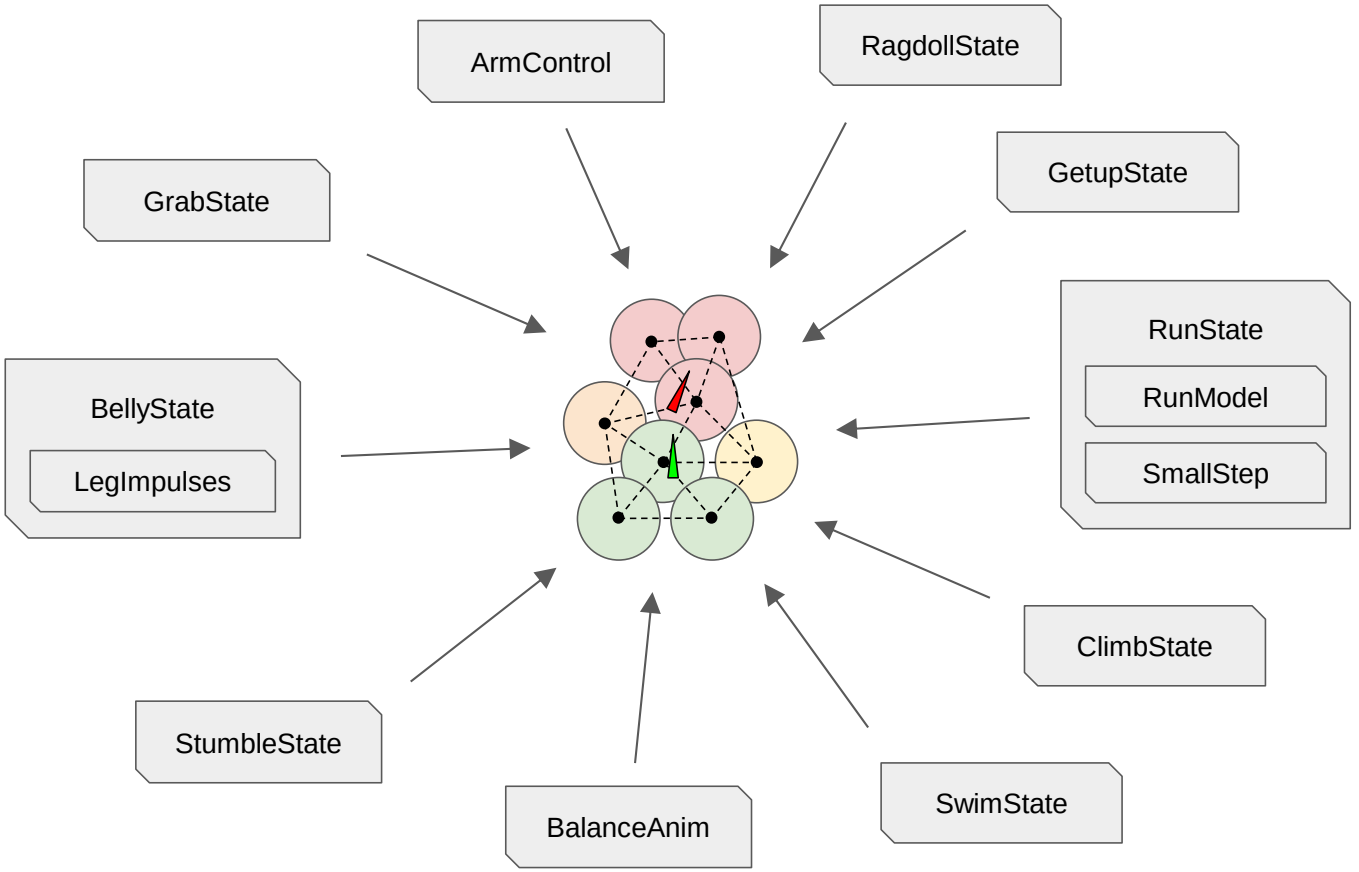
and more ...

A network of impulses



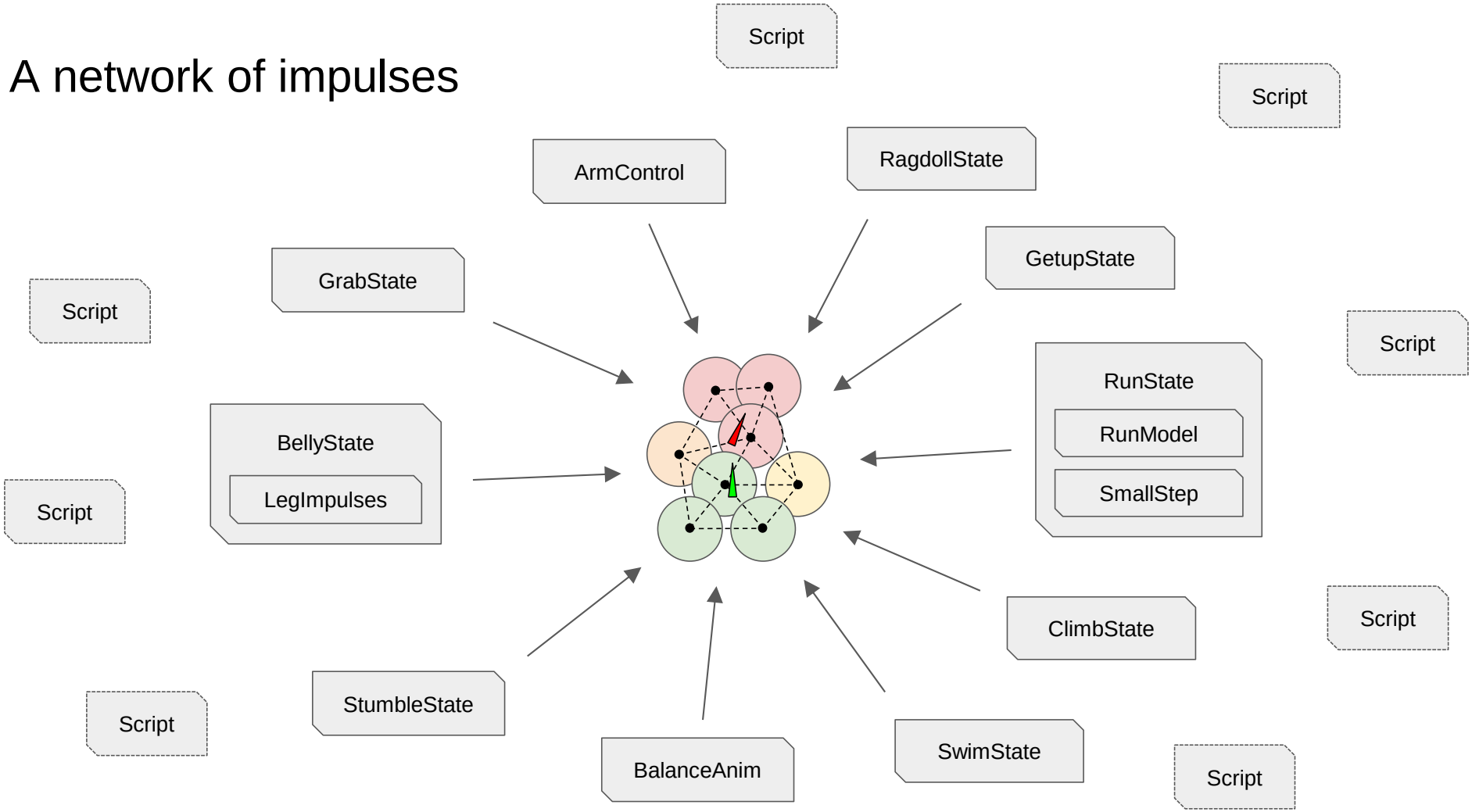
...

A network of impulses



and even more ...

A network of impulses



and auxiliary systems and level-specific scripts
--and all of them apply impulses to the core

because there are so many of them, every impulse is just a vote
--we had to tweak for feel by observing the expression of the total system

A network of impulses

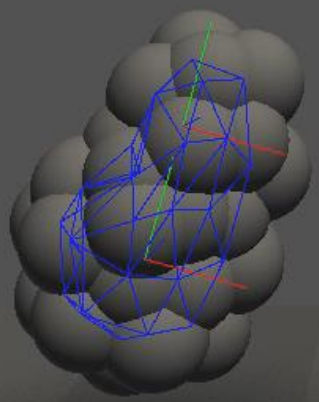
if you build something like this, then you get a system that is often surprising in how it behaves

there is an unmistakable perceived depth to the motion of the huddle, perhaps because it is so difficult to predict
--for better or worse, this is thanks to the underlying complexity of its many independent parts

I've talked a bit about the huddle core, but the core is just a small part of the total system
--next up is Søren, to tell us about its arms and legs ☺

Coming up: Animation Programming

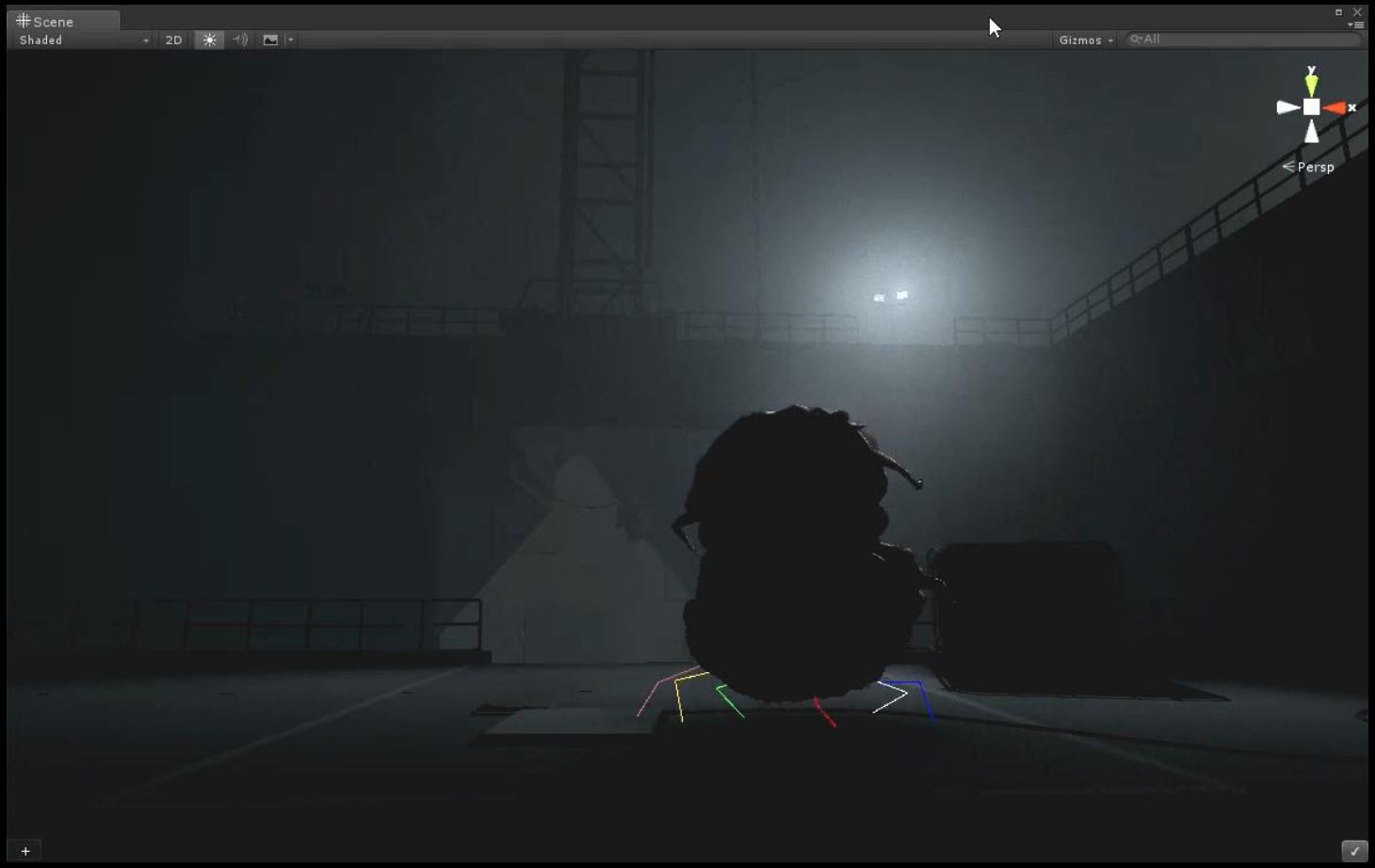
Søren Trautner Madsen



[long clip showing the huddle moving around, bridge to Søren’s segment]

This does not look heavy or physical or realistic, but it is the RELEASED version, just with invisible legs, moral: to be sure we were on the right track we needed those arms and legs added early.

Another example, hand drawn stick arms! It does not take much to completely change the perception.

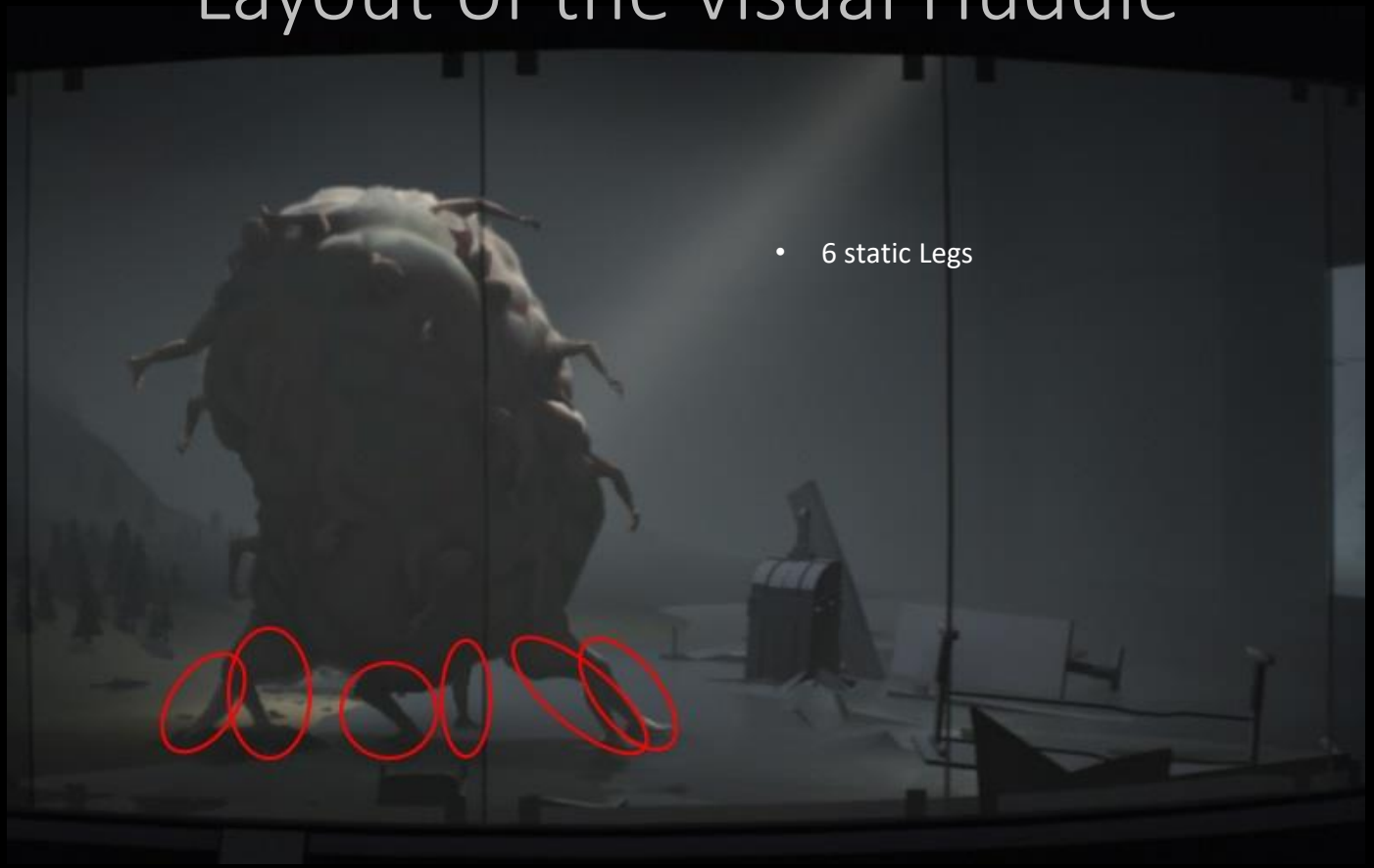


Do the same with the blob/huddle. Just these debug-lines are enough to feel the weight and start believing.

Layout of the visual Huddle

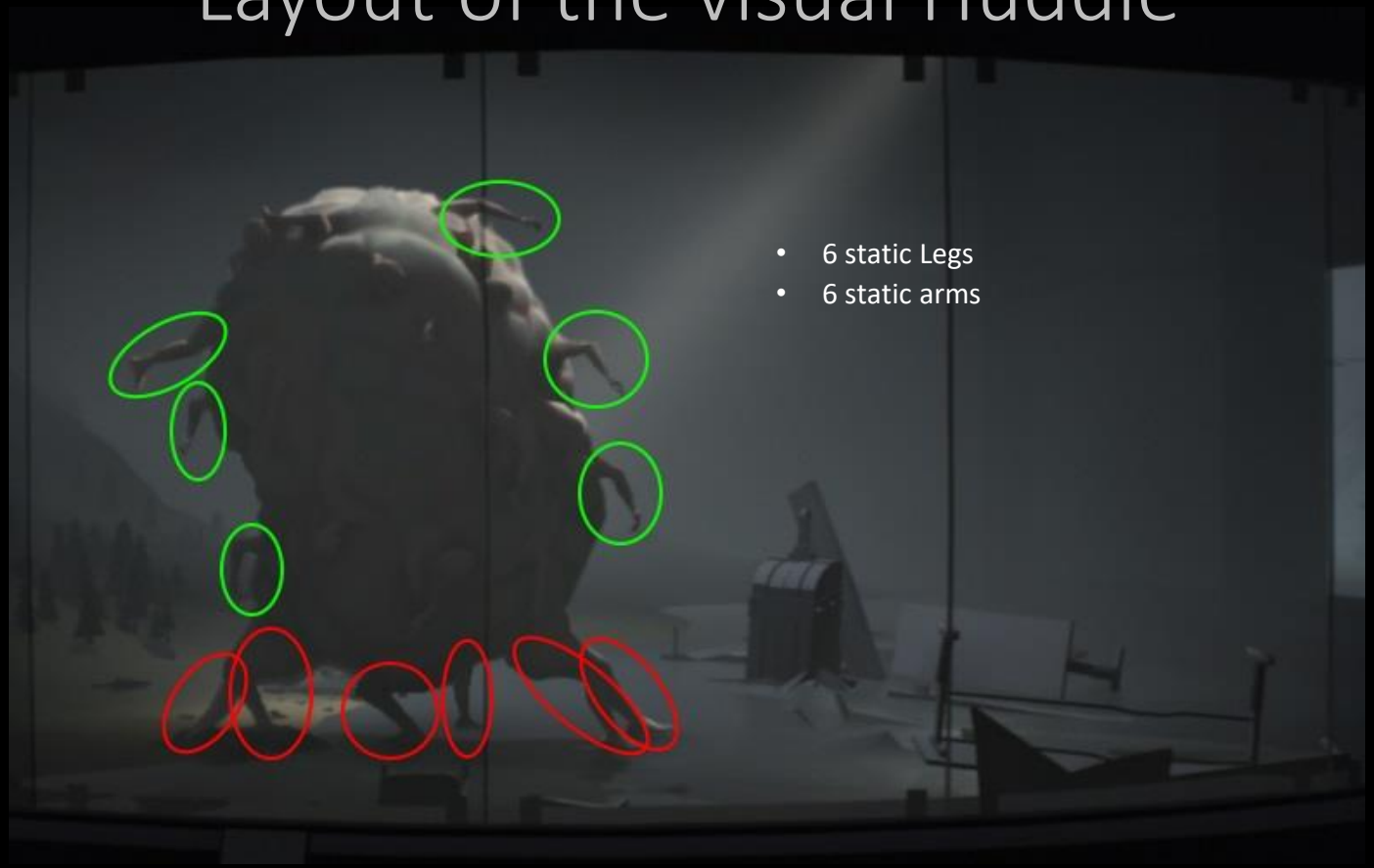
So this is the huddle in all its glory. The center part has been skinned with a mesh.

Layout of the visual Huddle



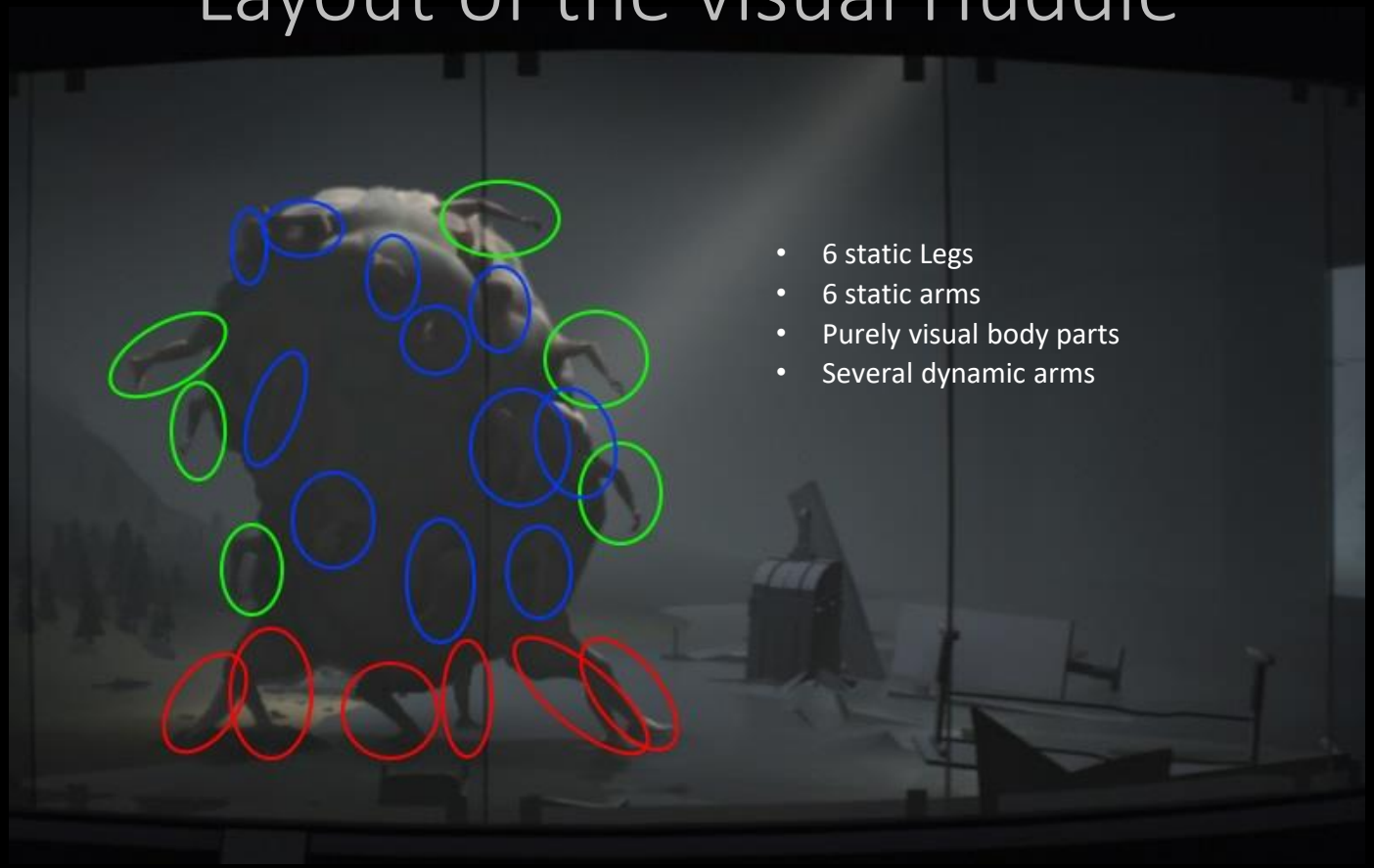
Placed on the lowest ”physics balls”. If the huddle falls and reconfigu.res the spine, these legs are allowed to shift to the new lowest ”physics balls”

Layout of the visual Huddle

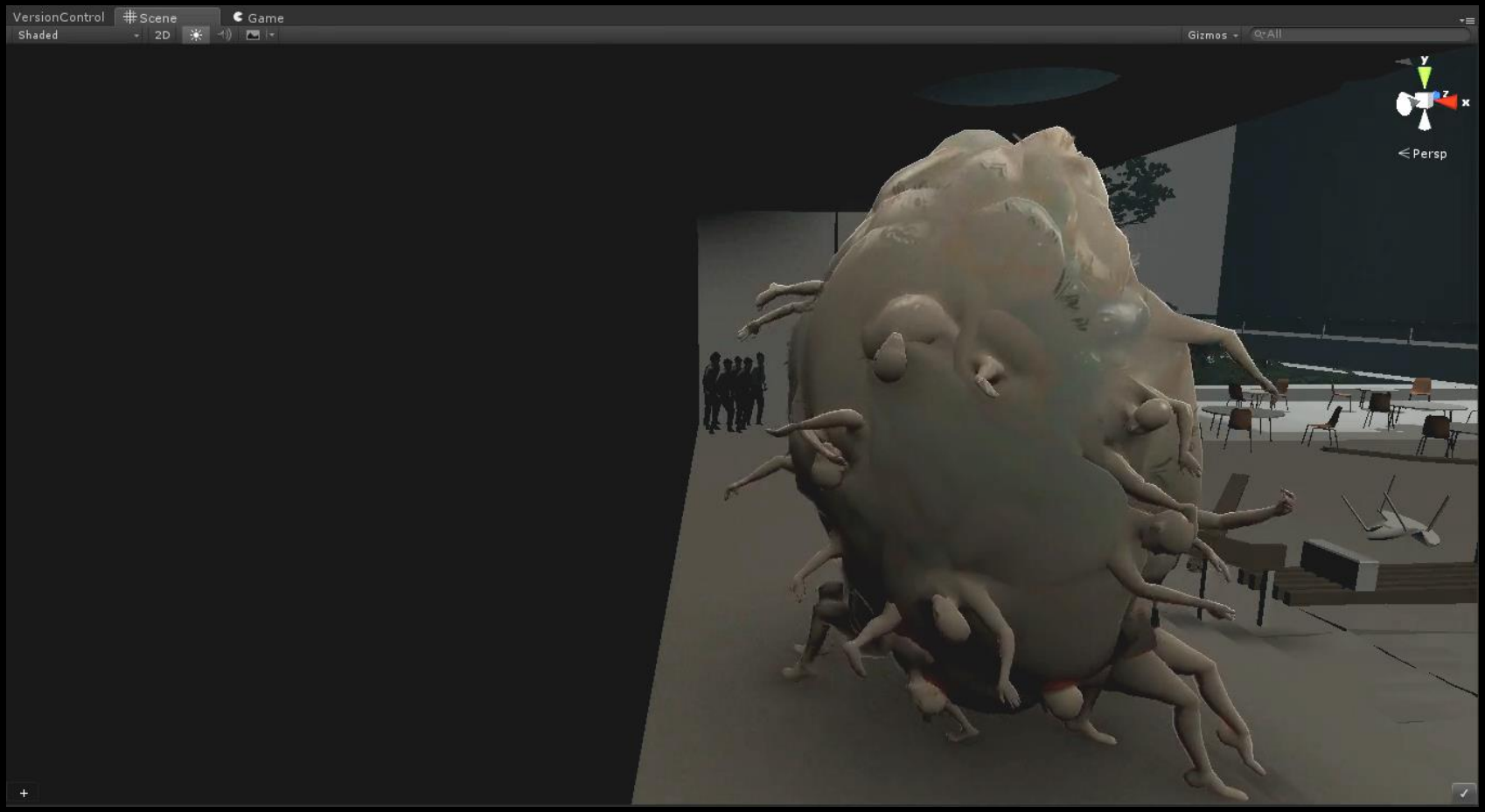


6 arms that are always placed nicely spread on the edge to give a good visual silhuette. These arms will touch objects in the scene and be able to grab boxes.

Layout of the visual Huddle

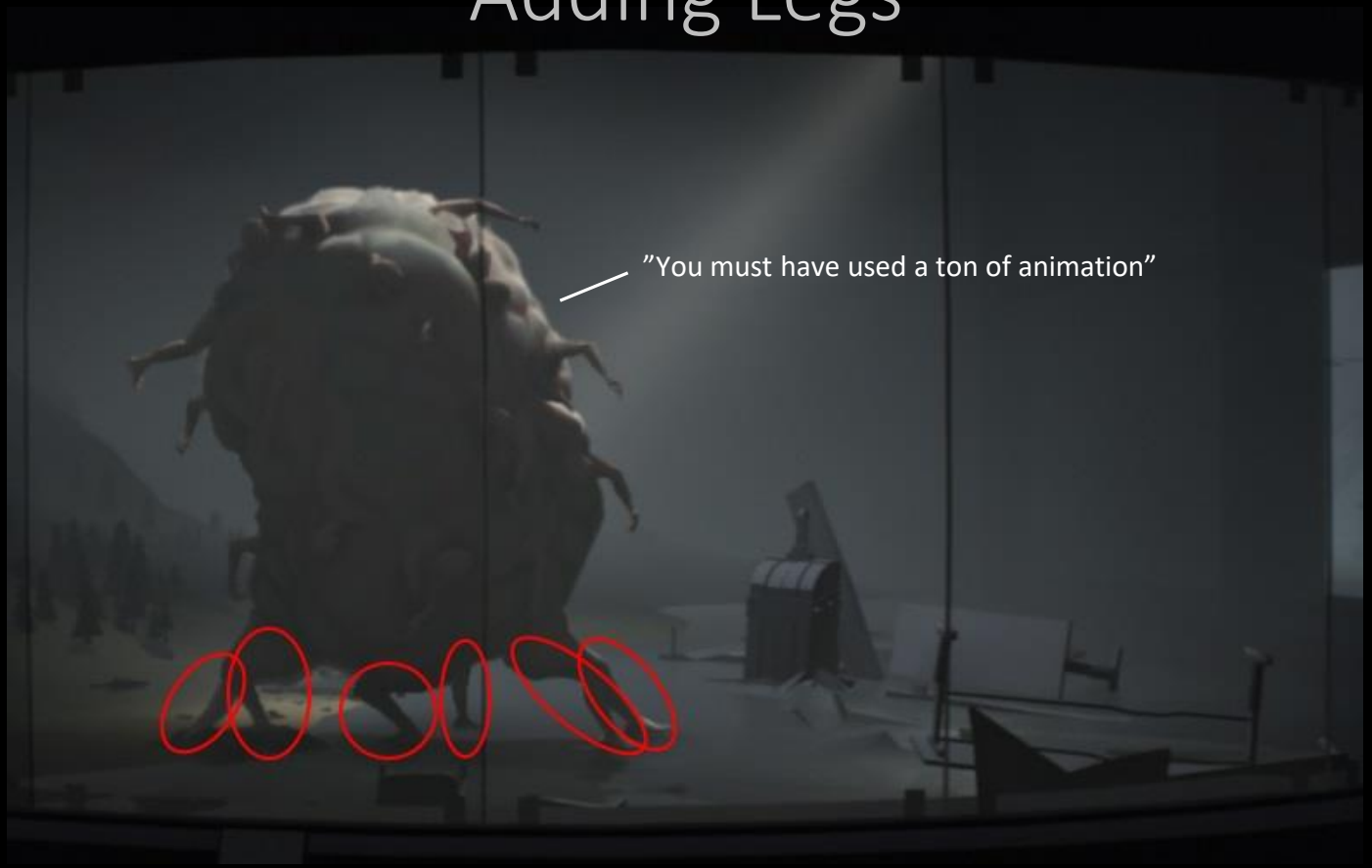


Visual body parts to give secondary motion and make the huddle look alive. Also a big pool of dynamic arms that can shoot out from the center of the huddle and assist with grabbing objects in the world (normally disabled)



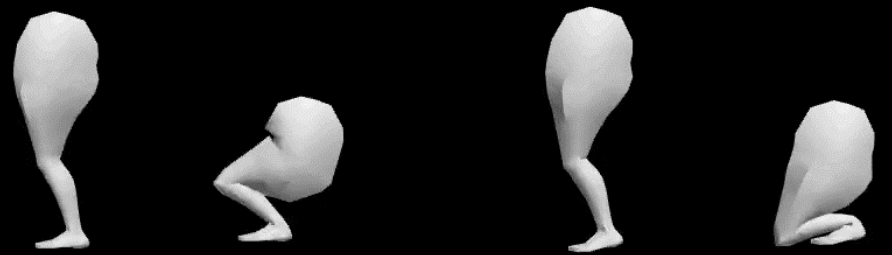
The purely visual parts. Initially glued un ragdolls. For Performance reasons ended as simple custom springs with no collisions. Forces added each frame to make them squirm and wiggle. To reduce clipping they pull back into the body if walls or other objects are "near".

Adding Legs

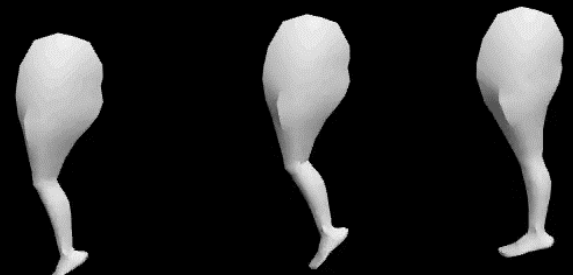


Since the huddle physics was constantly being developed, the design strategy was to just add a thin visual layer, that simply added on top of the existing simulation.

All Leg animations



Running



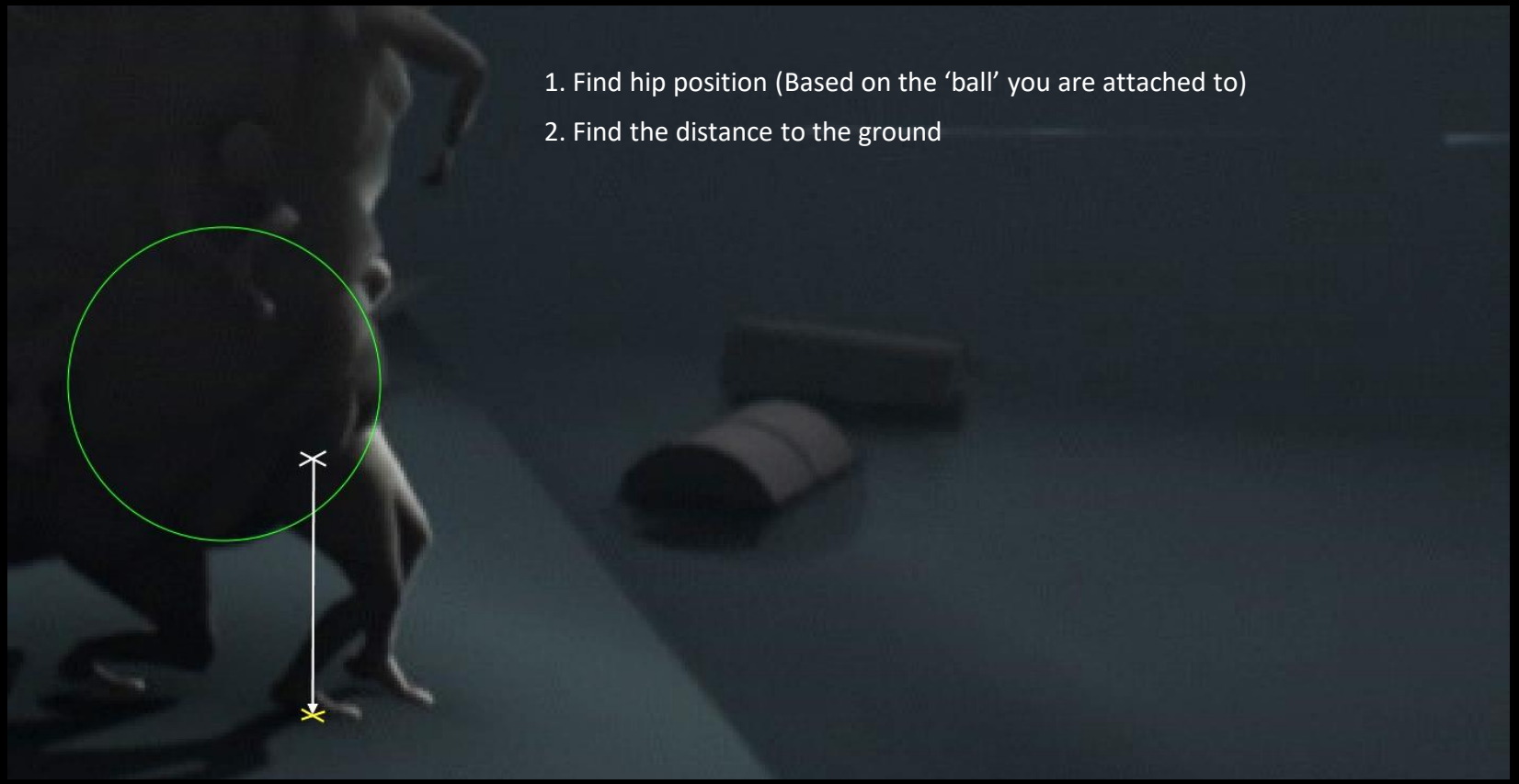
Dangling Falling

Every single leg animation seen in the game. On the top: 2 different pairs of running animations. Each pair has a “high” running animation and a “low” running animation. The top left pair is the normal running animation – the right is for running backwards. Further more at the bottom: An animation for a leg, that can not reach the ground, and 2 animations for when the huddle is falling.

Leg animation algorithm

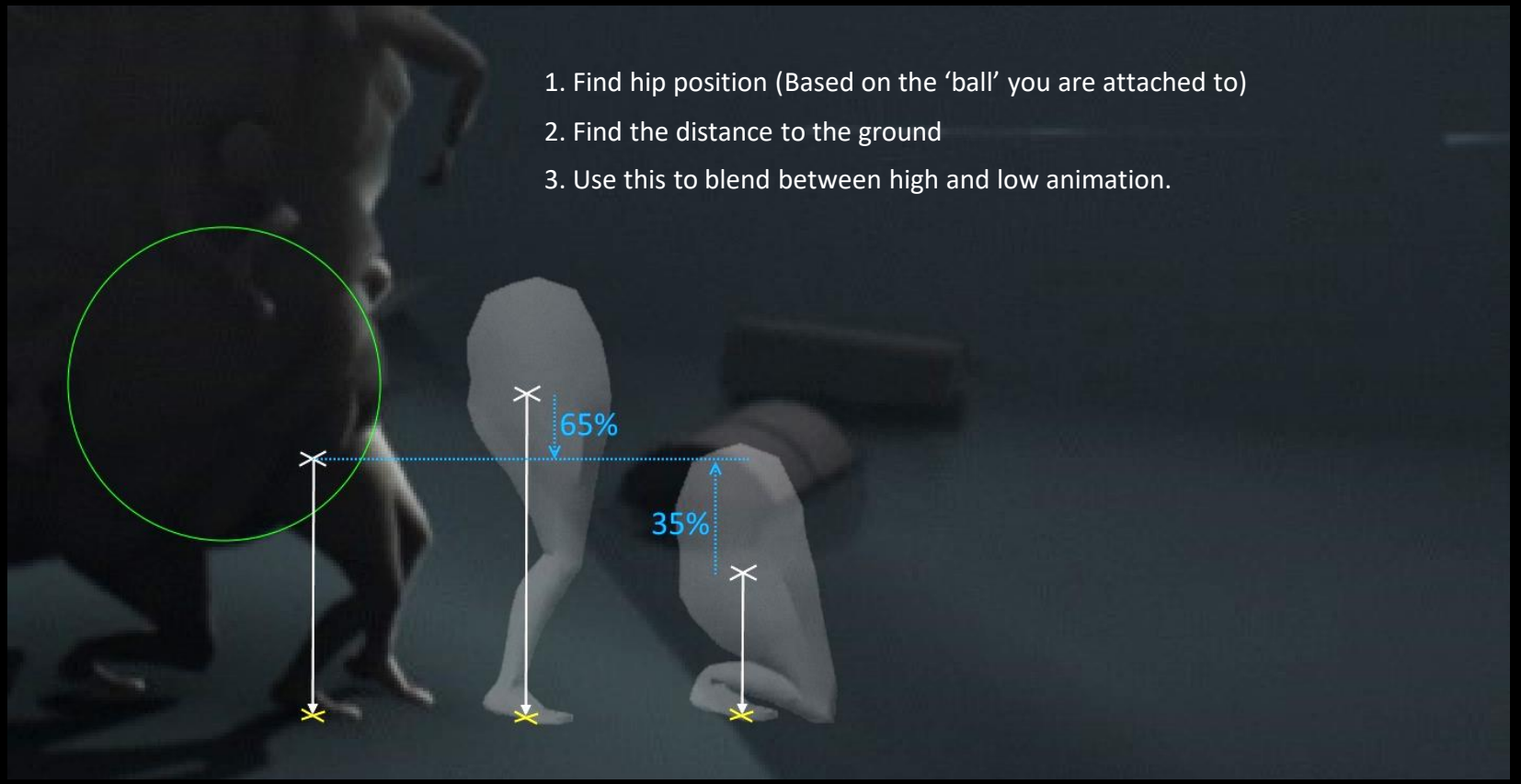
Each leg is attached to one of the physics balls at the bottom of the huddle – each of the 3 lowest balls have 2 legs attached

Leg animation algorithm



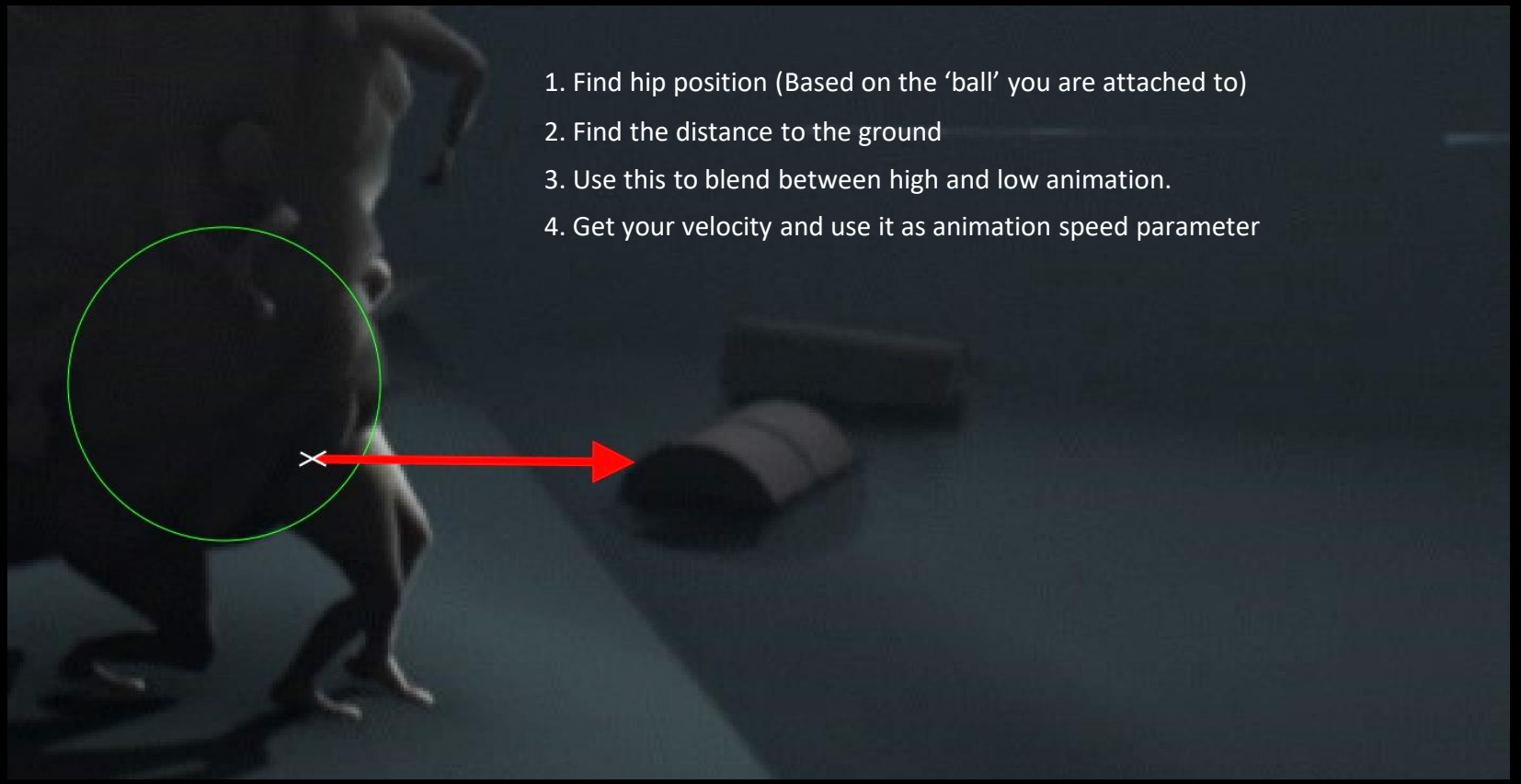
Shoot a ray from the ”hip position” of the leg to find the ground position.

Leg animation algorithm



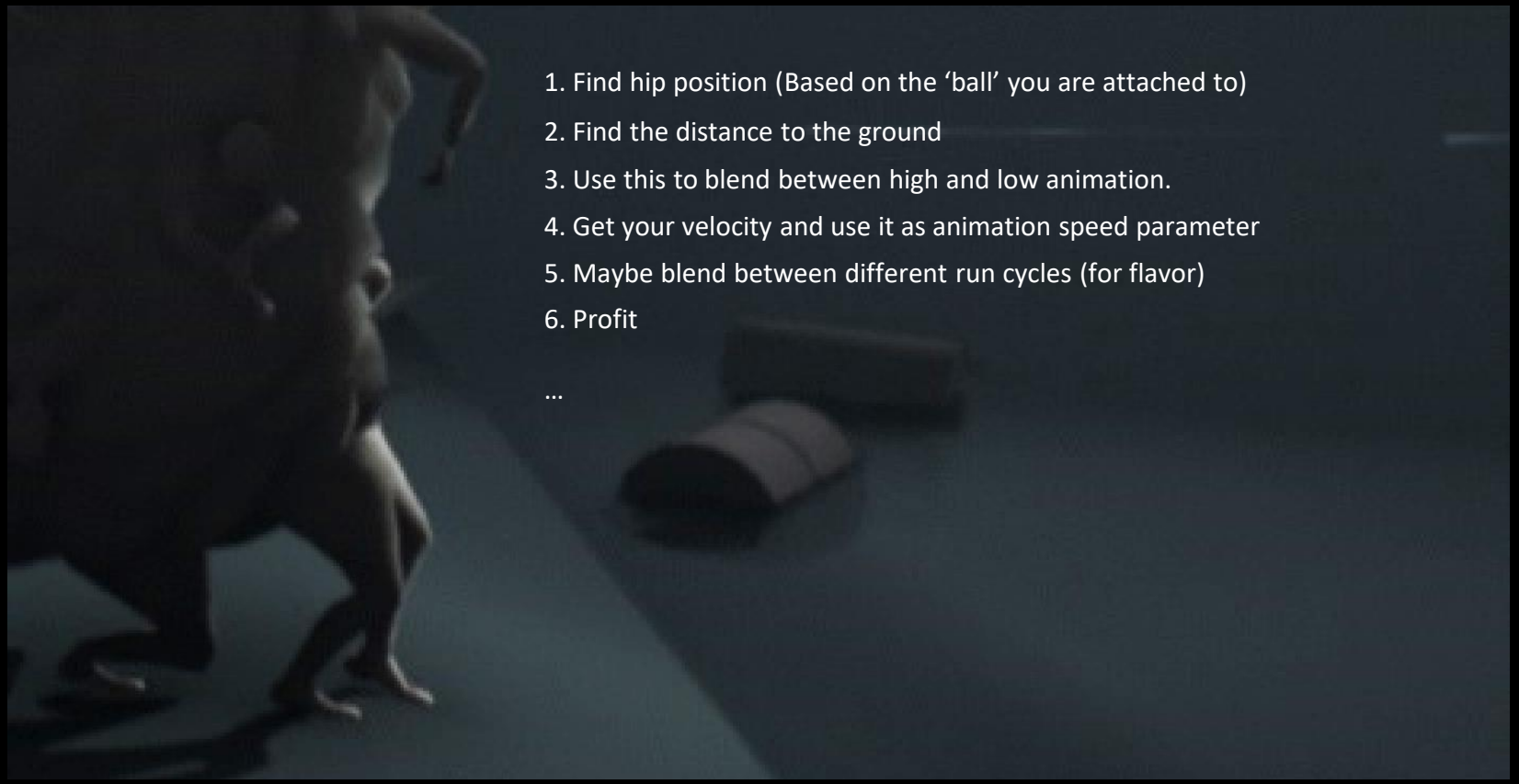
The distance from hip to ground is used to find a value used to blend between the high and low run animations. These animations are kept in a constant blended state.

Leg animation algorithm



We use the actual movement of the physics body to calculate the speed parameter of the leg animations. This way the leg animation will correspond to the actual movement speed, so the feet will not slide.

Leg animation algorithm

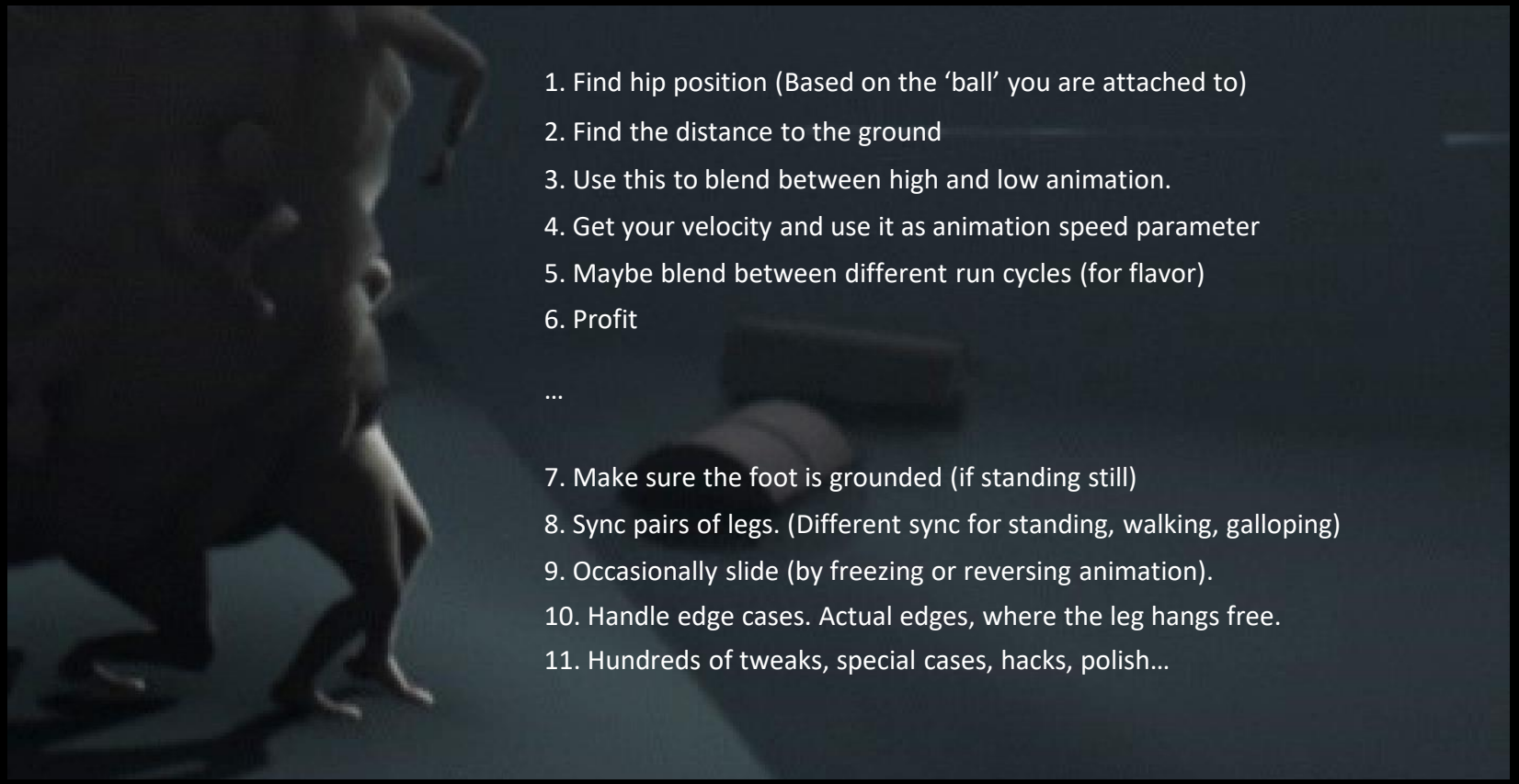


1. Find hip position (Based on the ‘ball’ you are attached to)
2. Find the distance to the ground
3. Use this to blend between high and low animation.
4. Get your velocity and use it as animation speed parameter
5. Maybe blend between different run cycles (for flavor)
6. Profit

…

And that’s it…
except of course – it isn’t

Leg animation algorithm



1. Find hip position (Based on the ‘ball’ you are attached to)
2. Find the distance to the ground
3. Use this to blend between high and low animation.
4. Get your velocity and use it as animation speed parameter
5. Maybe blend between different run cycles (for flavor)
6. Profit

…

7. Make sure the foot is grounded (if standing still)
8. Sync pairs of legs. (Different sync for standing, walking, galloping)
9. Occasionally slide (by freezing or reversing animation).
10. Handle edge cases. Actual edges, where the leg hangs free.
11. Hundreds of tweaks, special cases, hacks, polish…

7: The leg can be left in mid air when the huddle stops. We handle this by blending it into another frame, where it is on the ground.
8: When standing still the legs are both on the ground, pointing away. When walking the legs should be half a cycle apart. When gallopping they are about a quarter cycle apart.
9: We can set the wrong animation speed on purpose to make a leg slide on the ground (for instance if doing a hard turn, or the ground is slippery)

The animation leg blending. The big red arrow is the leg velocity. The white line is the distance to the ground (used for blending)

Animation on slippery floor. playing around with the animation speeds can make the floor slippery. The physics are (almost) the same - only the legs animation-velocities differ.

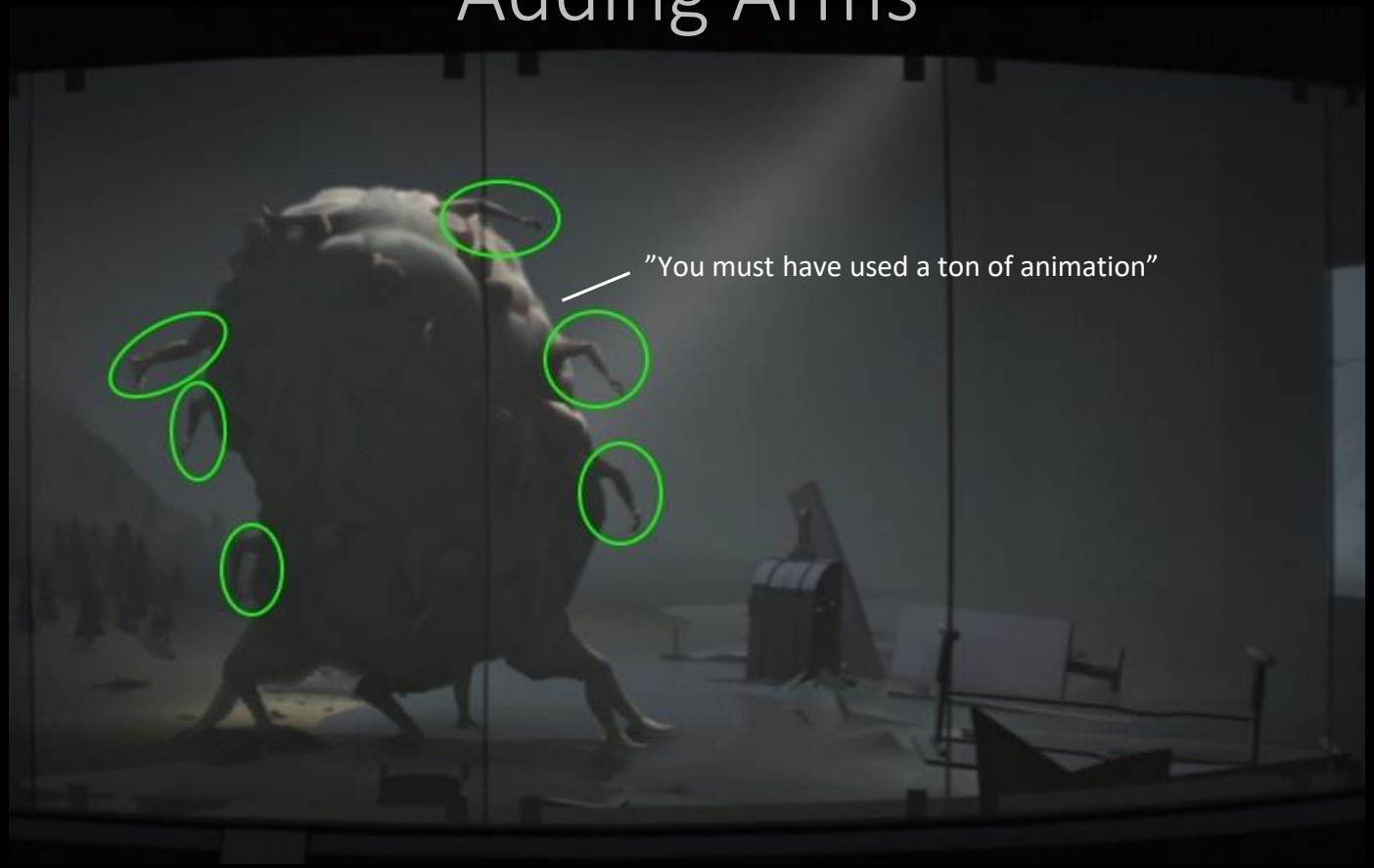
Force Feedback from Legs

No Feedback

Force Feedback

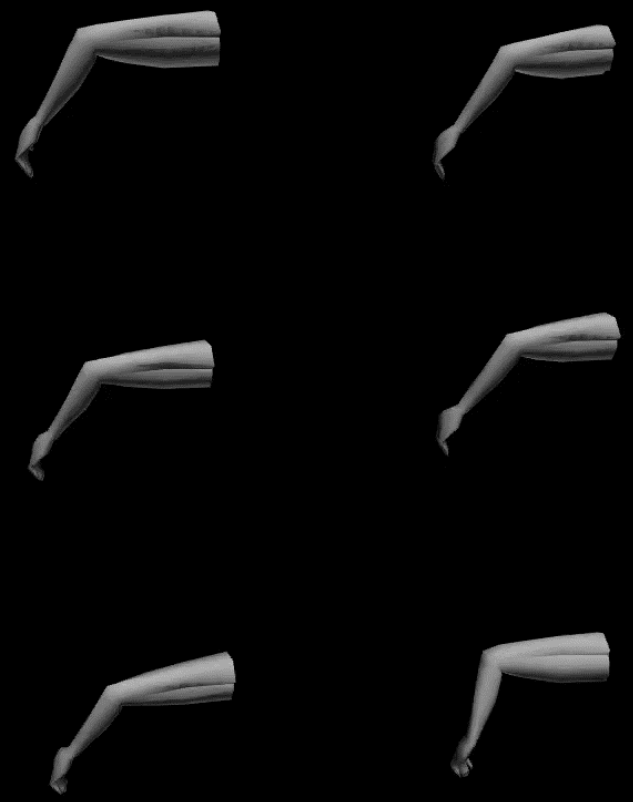
One of the few places I got to modify the physics from the visual overlay. Whenever a leg plants, it sends a wave of impulses up the body.
Other feedback from visualization to simulation:
Boxes can not be lifted until a lifting arm actually touches them
If an arm overstretches, the box is pulled towards the arm (And the bones towards the box)
Retract visual body parts (and their bones) from scene objects.

Adding Arms



The six static arms should be able to lift and touch things in the scene
They are not enough, so several dynamic arms can spawn at the center of the body and shoot out at an instances notice.

All Arm animations - Idles

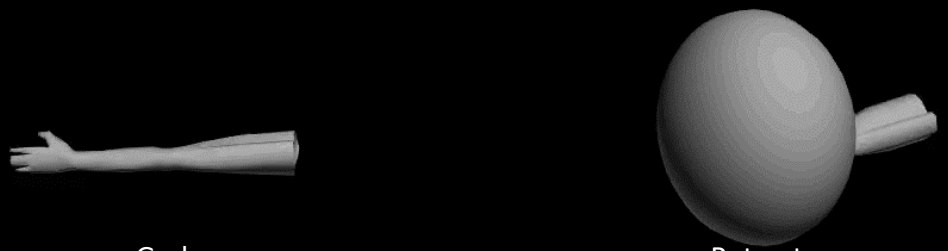


Just the idle animations for the static arms. These are spiced up by springs in the joints adding secondary motion

All Arm animations – Actions

Pain

Almost able to Reach



Grab

Retract

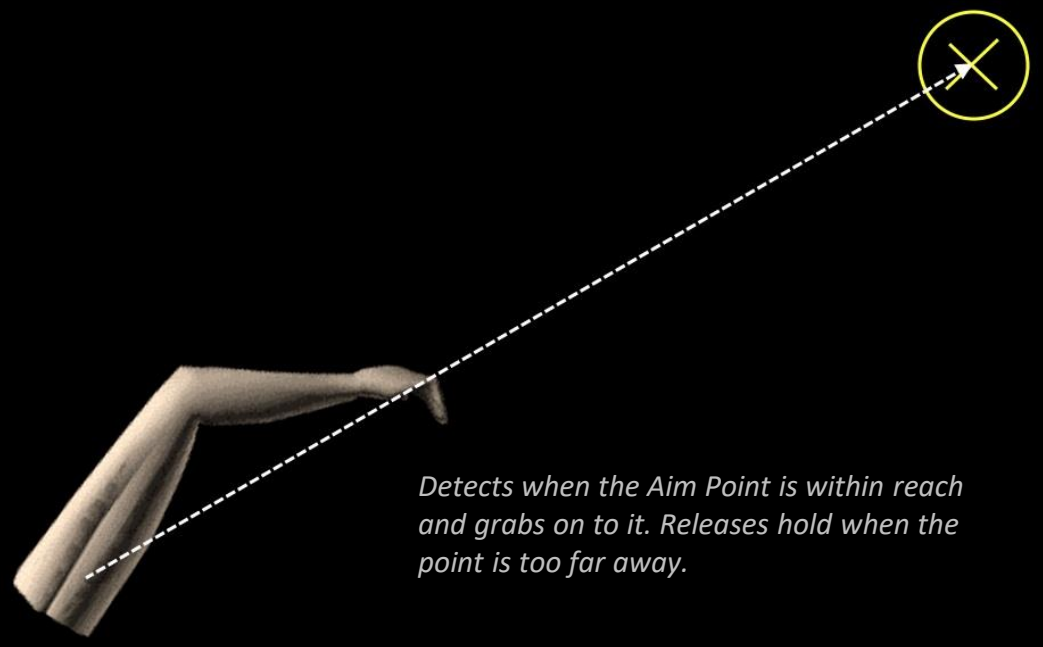
”Grab” and ”Retract” are the 2 animations that handles almost all gameplay.
The play position of the grab animation is set by code to have the hand at a specific distance to the shoulder. Inverse Kinematics is then used to get to position just right, and rotate the hand to align to the grabbed surface.

Grabbing with no arms

Grabbing with arms

Getting arms to grab

- Each arm aligns towards an “Aim Point”

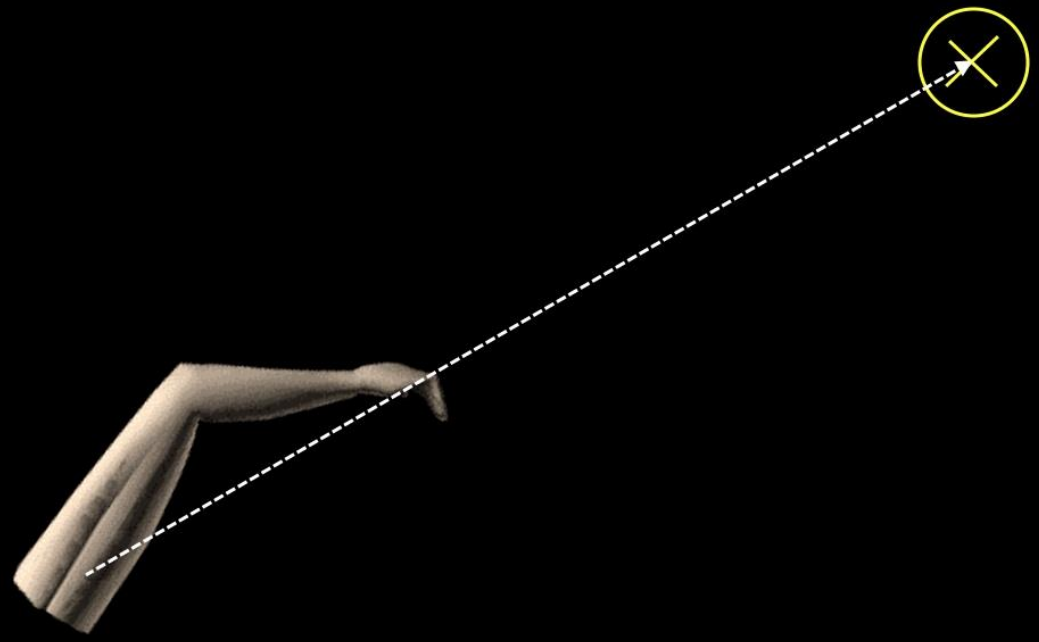


Autonomous Aim Point – responsible for making the arms point in the right directions.

Detects when the Aim Point is within reach and grabs on to it. Releases hold when the point is too far away.

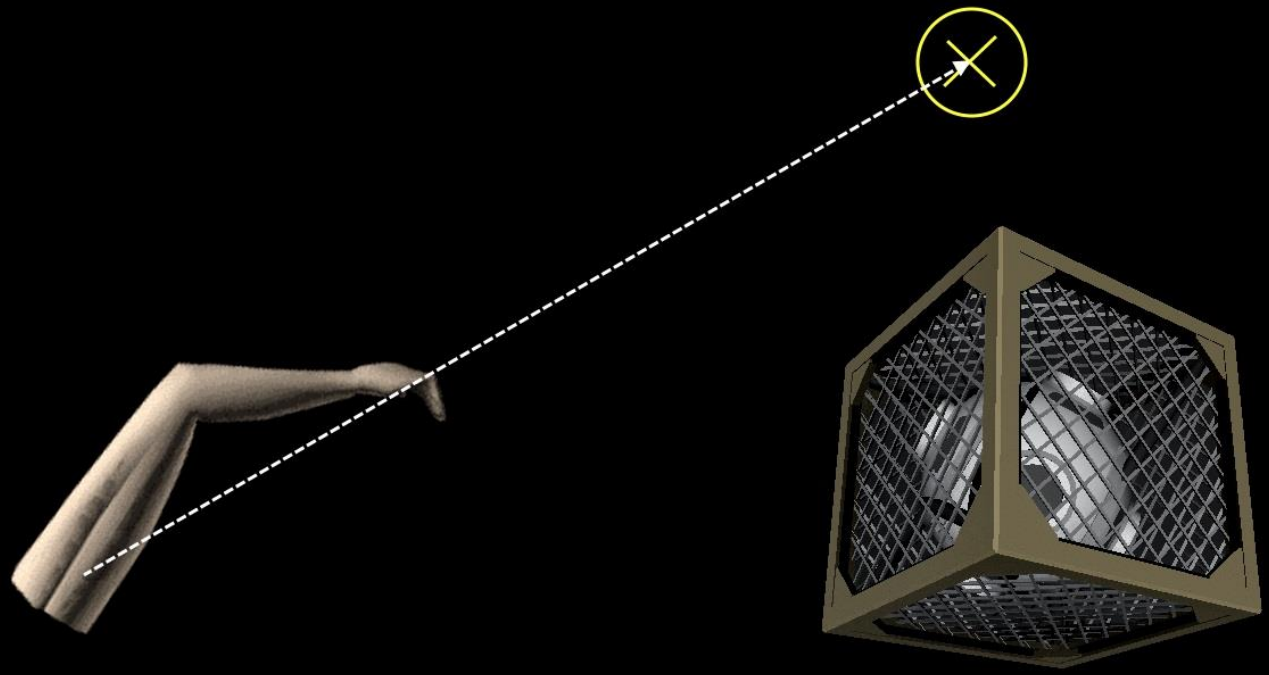
Getting arms to grab

- Each arm aligns towards an “Aim Point”
- An “Arm Manager” keeps track of all arms and interesting objects



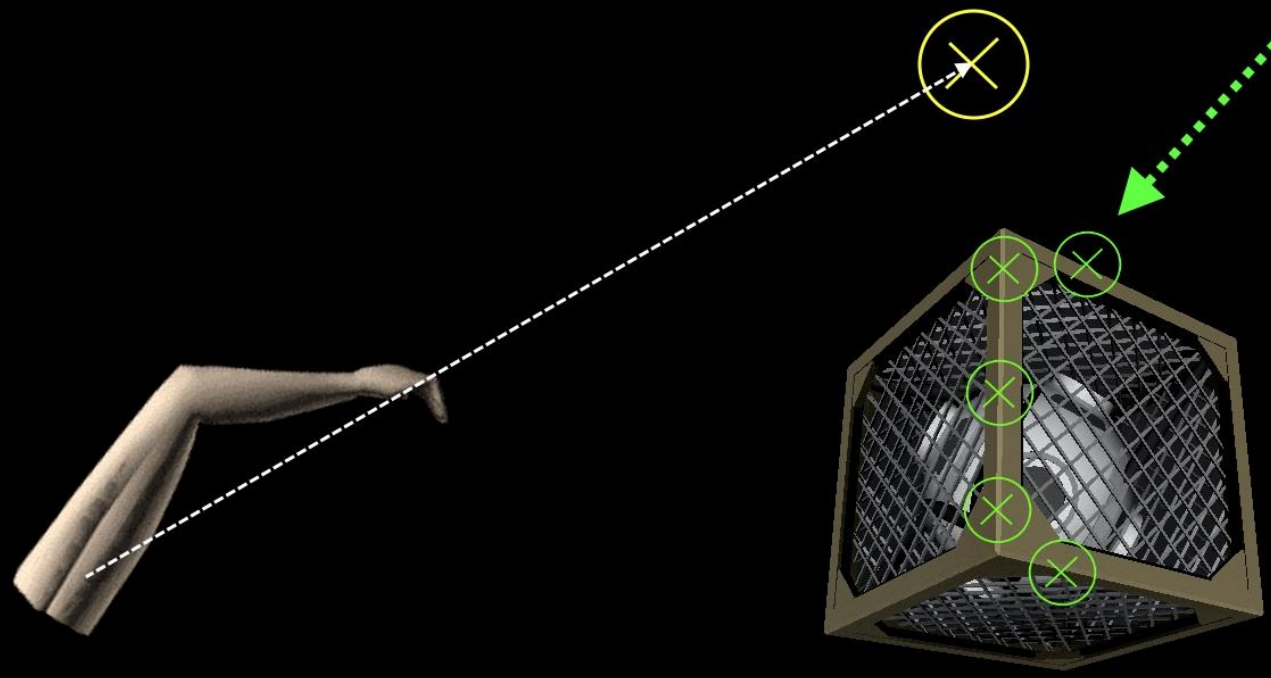
Getting arms to grab

- Each arm aligns towards an “Aim Point”
- An “Arm Manager” keeps track of all arms and interesting objects
- When the Huddle is about to grab something…



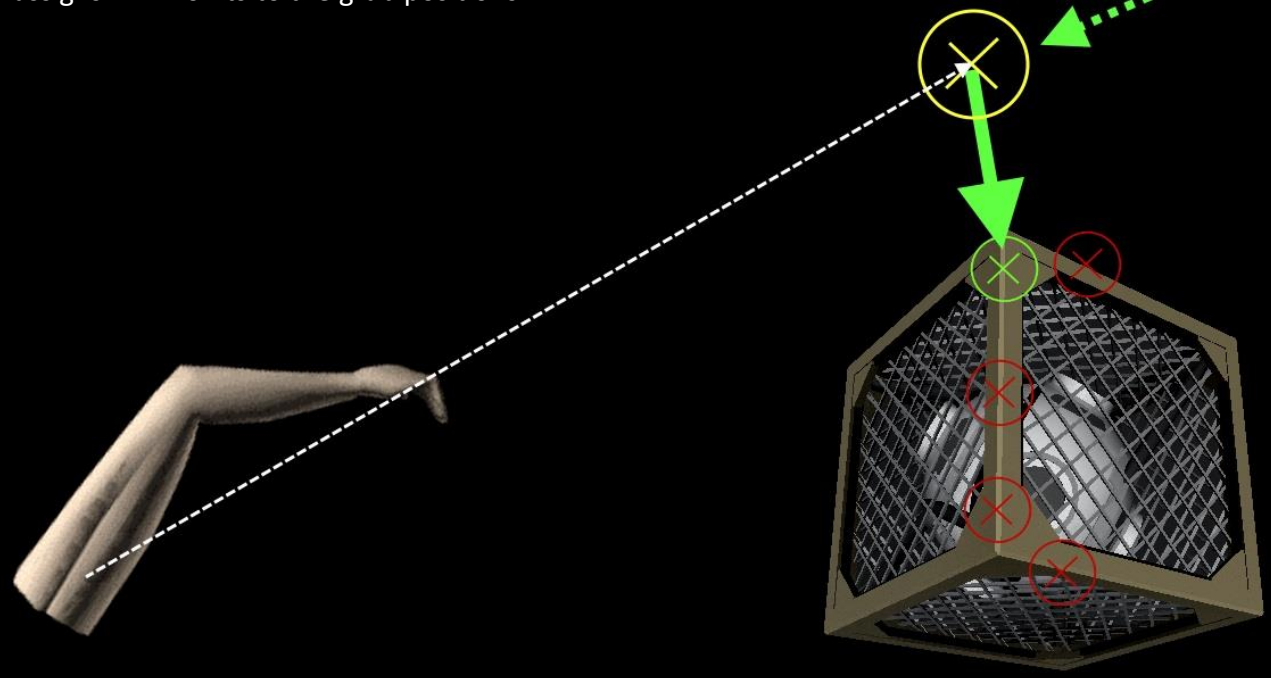
Getting arms to grab

- Each arm aligns towards an “Aim Point”
- An “Arm Manager” keeps track of all arms and interesting objects
- When the Huddle is about to grab something…
- the arm manager creates interesting grab positions…



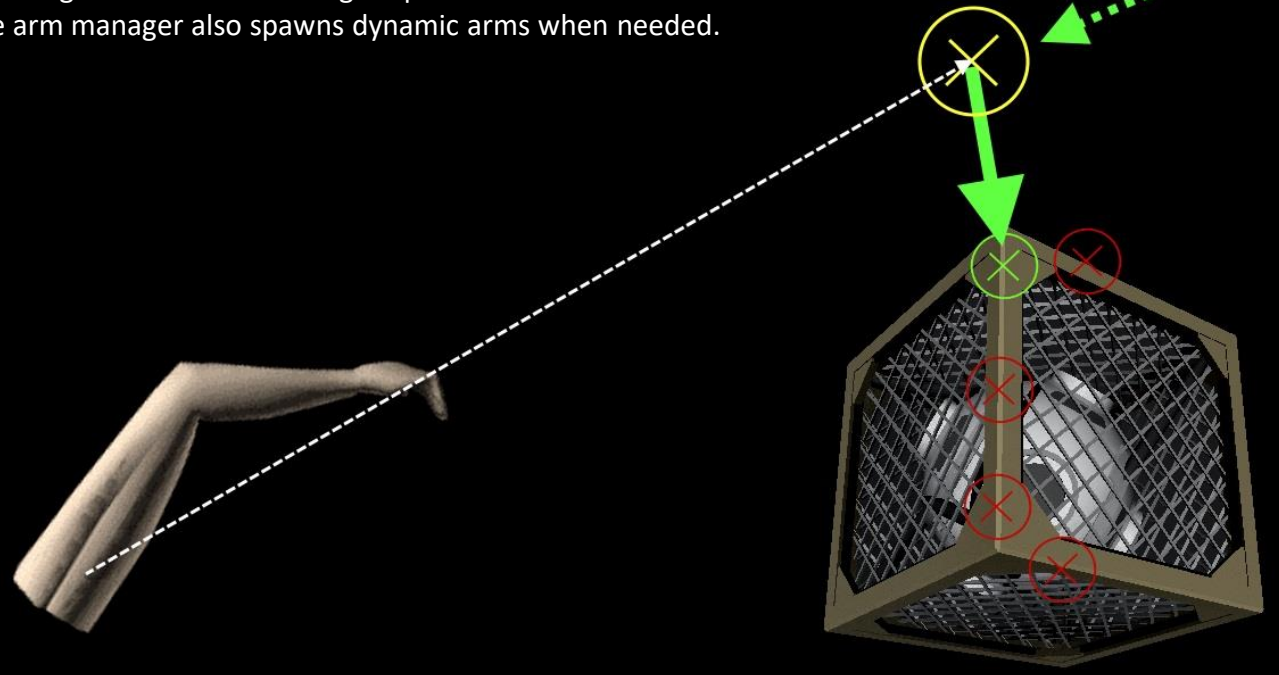
Getting arms to grab

- Each arm aligns towards an “Aim Point”
- An “Arm Manager” keeps track of all arms and interesting objects
- When the Huddle is about to grab something…
- the arm manager creates interesting grab positions…
- and assigns Aim Points to the grab positions



Getting arms to grab

- Each arm aligns towards an “Aim Point”
- An “Arm Manager” keeps track of all arms and interesting objects
- When the Huddle is about to grab something…
- the arm manager creates interesting grab positions…
- and assigns Aim Points to the grab positions
- The arm manager also spawns dynamic arms when needed.



At any point we want at least 2-3 arms holding on to a box. There are not enough static arms on the edge of the huddle for this. So dynamic arms are spawned from the center of the huddle and quickly shoot out to grab the box the same way as the static arms at the edge.
When one of these dynamic arms can no longer reach the box, the aim point will shoot away, and the arm will retract back to the center of the huddle.

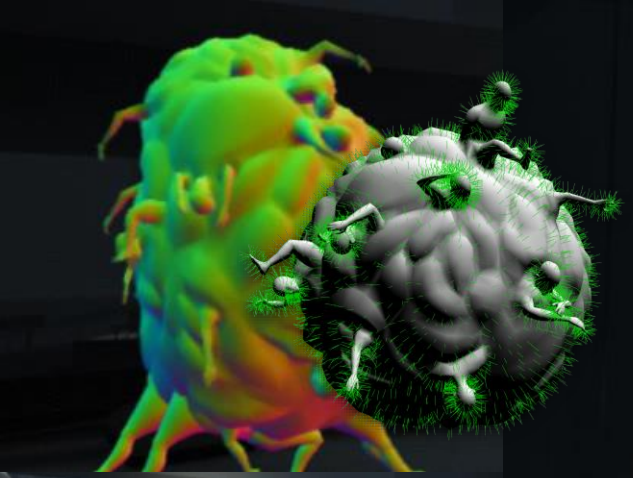
Climbing over stuff (in slow motion). The arm manager detects that a climb is occurring, it assigns a few arms to rest on the surface of the object being climbed. (also spawning new arms in the process)

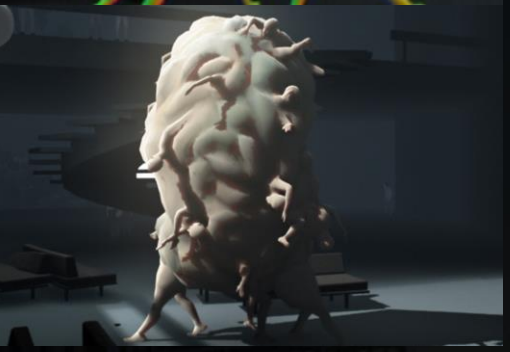
Grabbing a box (in slow motion) shows how the ToPoints are being assigned to the edge of the box. also shows how A LOT of dynamic arms are being spawned and killed

Hi, I'm Mikkel Svendsen, and on INSIDE I mostly did VFX like smoke, fire, and destruction, as well as making and maintaining water. But, for this thing I got to do something fun, namely guy's skin, or rather his surface appearance or shader.

So this is what it looks like, in the final game, no edits. During the 5 years it was made. we tried a bunch of things, Andreas tried some texturing stuff and some rotating ball thingies, Mikkel Gjøl had a round with the shader, and I had a couple as well. But, in the end it was time to settle on something, so I began on the final redo of the shader, a thus lot of people sent some inspiration my way...

I was sent John Isaacs sculptures, for the smooth, yet wrinkly and creasy shapes. Rembrandt paintings, for the oiliness and wide, yet muted palette, his using a lot of blue, green, yellow and generally unexpected tones for skin. And Jenny Saville's paintings, right in the middle, they've got a bit of the best of both worlds for us. Apart from looks, there were some issues, like Søren mentioned, the Huddle is composed not only of the 26 physics balls that Lasse talked about, but also a bunch of dynamic auxiliary limbs, and this all had to look like more than just the sum of it's parts, so some shader glue had to be utilized...

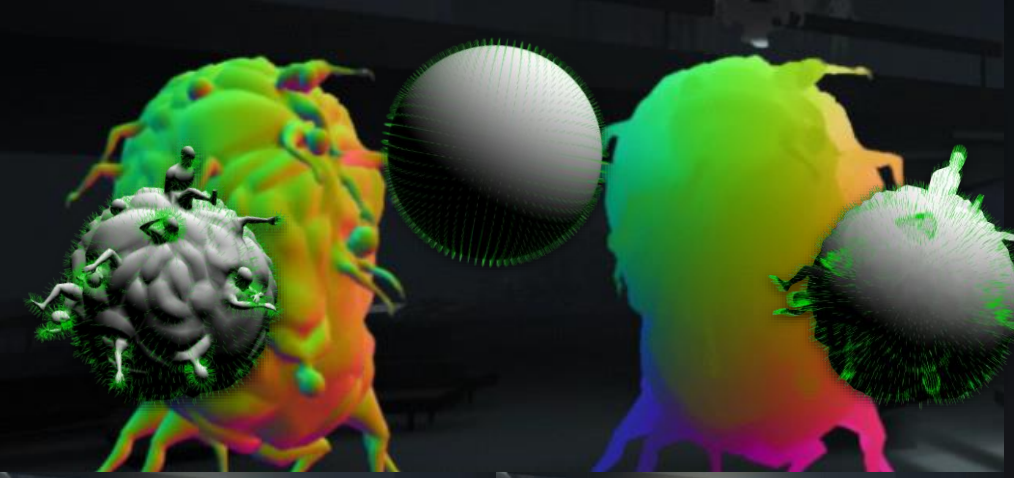


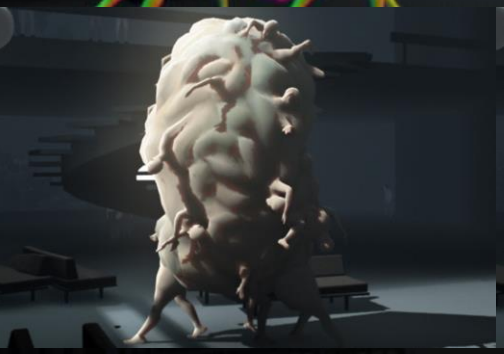


"Hard" Model's Normals

```
//Normals from vertex data
float3 hard = objectToWorldDir(oNorm);
```

Let's start with the most visible stuff, normals. This is how the normals look with no effects on, it's just the surface derivative normals from the model Andreas made, but it's got some issues, it's too noisy, and the limbs don't look very attatched, so we need to do something about that. You might think a Normal map is what I want, but no I want less detail, not more, so let's try some normal unmapping!





"Hard" Model's Normals

```
//Normals from vertex data
float3 hard = objectToWorldDir(oNorm);
```

"Soft" Ellipsoid Normals

```
//World space position
float3 pos = objectToWorldPoint(oPos);

//Direction to center, scaled, normalized
float3 ellipsoid = (pos - center) / size;
float3 soft = normalize(ellipsoid);
```

There! Done! No... So these "Soft" normals are simply done by pretending the Huddle is a sphere, by using the local position as normal. This fixes the legs, and gets rid of the noise, but all detail is lost, so now it looks like flesh soap, we need a middle ground, a blend factor...

"Hard" Model's Normals

```
//Normals from vertex data
float3 hard = objectToWorldDir(oNorm);
```

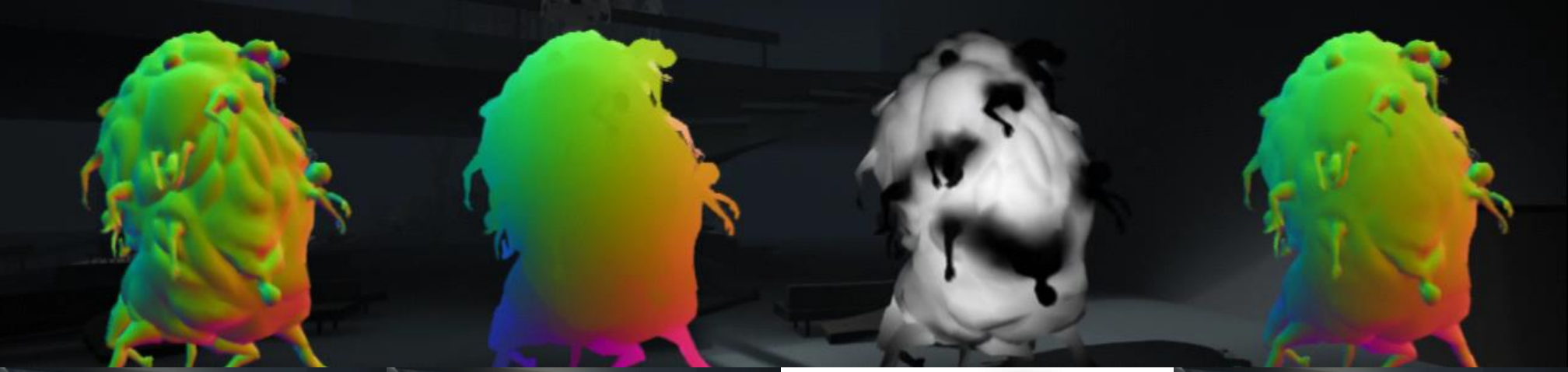
"Soft" Ellipsoid Normals

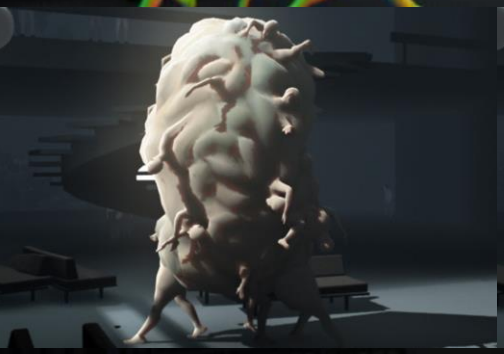
```
//World space position
float3 pos = objectToWorldPoint(oPos);

//Direction to center, scaled, normalized
float3 ellipsoid = (pos - center) / size;
float3 soft = normalize(ellipsoid);
```

Blend Factor

```
//Angle between both soft and hard normals
float dotHard = dot(view, hard);
float dotSoft = dot(view, soft);
//Bodies poking out and dynamic limbs
float bodies = tex2D(bodyTex, uv);
float limbs = tex2D(limbTex, uv);
//Mix it all together kinda like so
float blend = sqrt(dotHard * dotSoft);
blend = lerp(blend * bodies, 1.0, limbs);
```

So this is the blend we use, it's favoring the softness in the middle to keep it vauge, and tries to keep the detail around the edges and the rim. It also wants the detail to stay on Søren's spring bodies, but most notably the dynamic auxiliary arms and legs are fading from the soft to the hard from the shoulders and thighs to the tops of their toes and fingers...





"Hard" Model's Normals

```
//Normals from vertex data
float3 hard = objectToWorldDir(oNorm);
```

"Soft" Ellipsoid Normals

```
//World space position
float3 pos = objectToWorldPoint(oPos);

//Direction to center, scaled, normalized
float3 ellipsoid = (pos - center) / size;
float3 soft = normalize(ellipsoid);
```

Blend Factor

```
//Angle between both soft and hard normals
float dotHard = dot(view, hard);
float dotSoft = dot(view, soft);
//Bodies poking out and dynamic limbs
float bodies = tex2D(bodyTex, uv);
float limbs = tex2D(limbTex, uv);
//Mix it all together kinda like so
float blend = sqrt(dotHard * dotSoft);
blend = lerp(blend * bodies, 1.0, limbs);
```

Blended Normals

```
//Blend the results
float3 norm = lerp(hard, soft, blend);
```

That's it! This mix smooths out the rough edges, but retains the details we like. It also fixes legs/arms intersections nicely. So that takes care of normals, let's move on to lighting, a lot of lighting...

Just Ambient, No Occlusion

```
//Ambient light is just a parameter color
float ao = 1.0;
light += ambientColor * ao;
```

Let's start from the bottom, ambient. Like the boy, the Huddle has a fake ambient color, that's always there, so we can see it in the dark. And like the boy, it needs occlusion, but much more so, here it looks extremely flat, so let's add some!

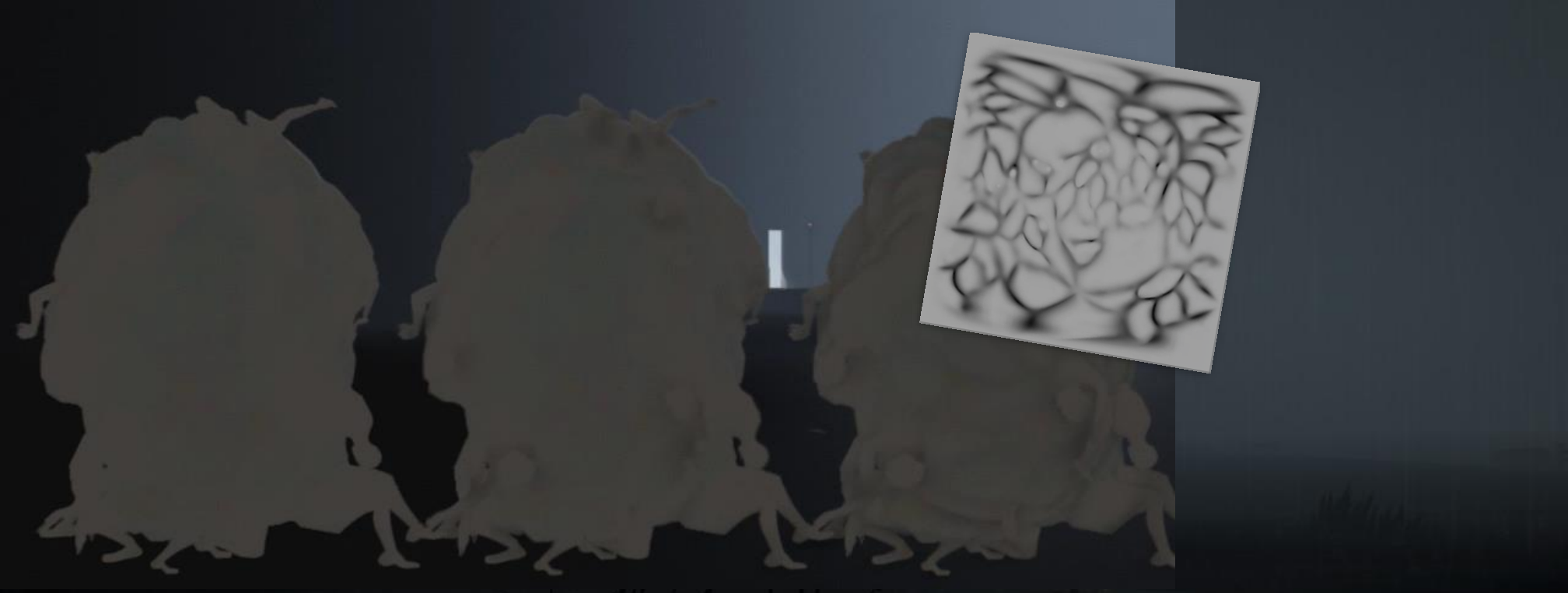
Just Ambient, No Occlusion

```
//Ambient light is just a parameter color
float ao = 1.0;
light += ambientColor * ao;
```

Limbs & Body Decals

```
//Just a bunch of decals that darken stuff
//Has nothing to do with Huddle's shader
//But naturally contributes
```

Decals! They cover the body parts, like arms, legs and the spring bodies. It's done using projected decals, just like the boy and NPCs like albinos and zombies. These decals are not actually part the huddle shader, it's a separate entity, you can think of it as a point light that removes light rather than add it.



Just Ambient, No Occlusion

```
//Ambient light is just a parameter color
float ao = 1.0;
light += ambientColor * ao;
```

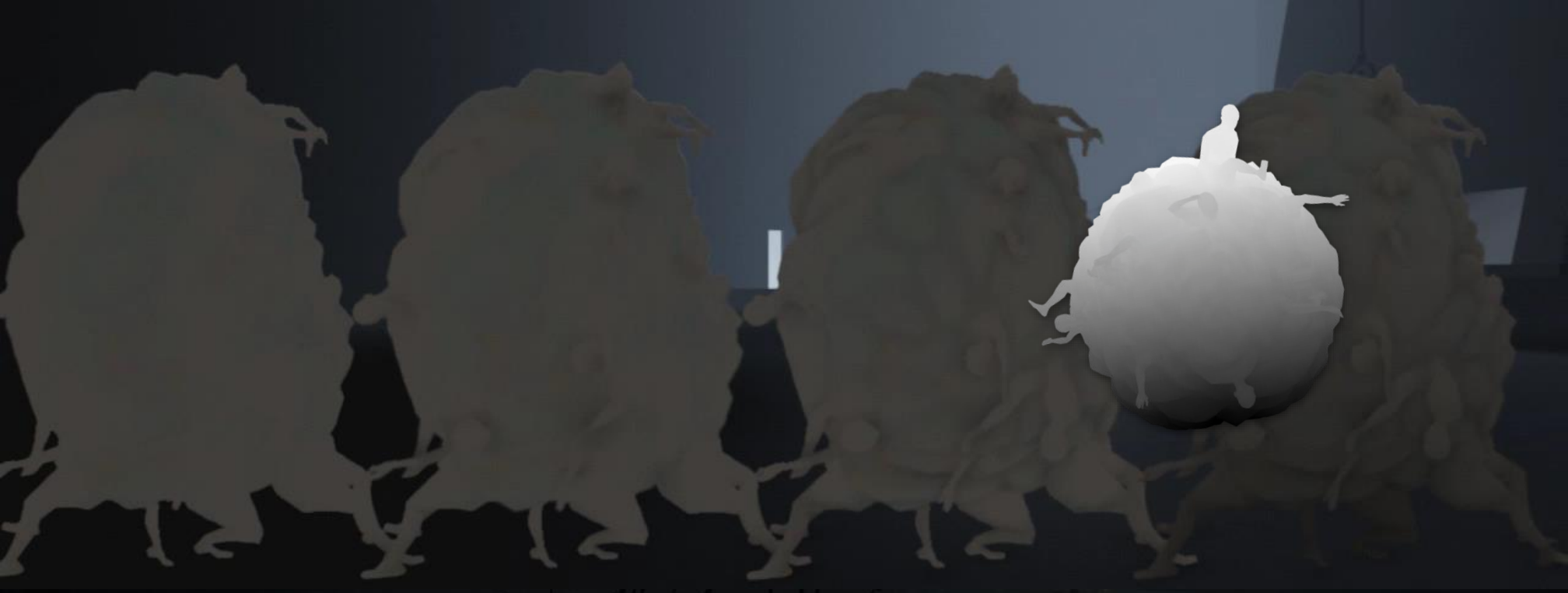
Limbs & Body Decals

```
//Just a bunch of decals that darken stuff
//Has nothing to do with Huddle's shader
//But naturally contributes
```

Baked Crease Texture

```
//Simple baked texture with body creases
//Multiply the ao by a sample of this
float aoTex = tex2D(aoMap, uv);
ao *= aoTex;
```

More prominently, we have a baked ao texture, very simple thing baked out from Max, using Andreas mesh with some tessellation. The only special thing is that I lightened it up in the center, while retaining detail at the rim



Just Ambient, No Occlusion

```
//Ambient light is just a parameter color
float ao = 1.0;
light += ambientColor * ao;
```

Limbs & Body Decals

```
//Just a bunch of decals that darken stuff
//Has nothing to do with Huddle's shader
//But naturally contributes
```

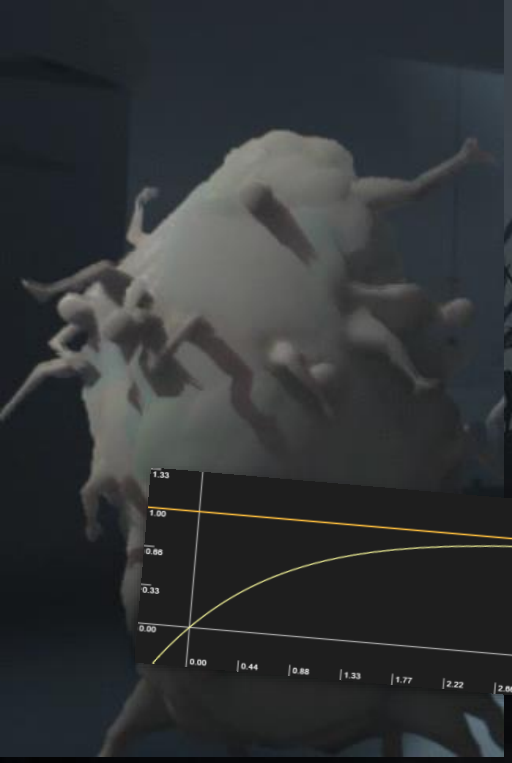
Baked Crease Texture

```
//Simple baked texture with body creases
//Multiply the ao by a sample of this
float aoTex = tex2D(aoMap, uv);
ao *= aoTex;
```

Upward Normal Gradient

```
//A weighted average of two normal
//Some of the hard, some of the soft
float aoGrad = 1.0;
float softness = soft * 0.35 + 0.65;
aoGrad *= lerp(hard, soft, softness);
aoGrad = aoGrad * 0.4 + 0.6;

ao *= aoGrad * (2.0 - aoGrad);
```

Finally, a vertical gradient, which is the most important. It makes it darker on the bottom while retaining the brightness on the top. This is sort of a self-occlusion thing that we don't use elsewhere, but since this is a big occlude it's quite important for it to look like it's on the ground. It's a very simple, stupid mix of the soft and hard y normals, but it's still just a vertical gradient at it's core.



Exponential Tonemapping

```
//Light stored in buffer as exp2(-light)
float3 lightBuf = tex2D(_LightBuffer, uv);

//Invert to get tonemapped lighting info
float3 light = 1 - lightBuf;
```

Next up in the lighting hierarchy, diffuse! So like almost everywhere in game, It's lambert with a bunch of tricks. The huddle couldn't do with just standard Lambert, most importantly, it mustn't burn out in bright light, so we tonemap it! We used exponentially decaying tonemap, simply because it was cheap... Actually it's free, cheaper than free, since we're rendering in Unity's LDR, rather than HDR "pipeline" which stores light in a multiplicatively blended, exponentially mapped buffer, so to get that tone mapping, we simply just don't unmap it.

Exponential Tonemapping

```
//Light stored in buffer as exp2(-light)
float3 lightBuf = tex2D(_LightBuffer, uv);

//Invert to get tonemapped lighting info
float3 light = 1 - lightBuf;
```

Thin Depth Offset Rimlight

```
//Offset screenspace uv by a few pixel
//along view norm, sample depth at offset
float2 uvOfs = uv + vNorm.xy * rimWidth;
float zOfs = tex2D(_DepthBuffer, uvOfs);

//Get delta from regular frag depth
float rim = saturate(fragPos.z - zOfs);

//Apply rim to lighting with parameter
light += rim * light * rimFact;
```

Then a little rim, like the boy and all the other NPCs. It helps them fit together, and helps it stand out, not that it needs it. Done with depth edge detection from the depth, and depth with a normal offset, which you might call a Sobel edge. It just amplifies the incoming lighting, simple as that.

Exponential Tonemapping

```
//Light stored in buffer as exp2(-light)
float3 lightBuf = tex2D(_LightBuffer, uv);

//Invert to get tonemapped lighting info
float3 light = 1 - lightBuf;
```

Thin Depth Offset Rimlight

```
//Offset screenspace uv by a few pixel
//along view norm, sample depth at offset
float2 uvOfs = uv + vNorm.xy * rimWidth;
float zOfs = tex2D(_DepthBuffer, uvOfs);

//Get delta from regular frag depth
float rim = saturate(fragPos.z - zOfs);

//Apply rim to lighting with parameter
light += rim * light * rimFact;
```

Wide Rim aka Roughness

```
//Dot product between view dir and normal
//Then tone it down with parameter
float rough = saturate(dot(view, norm));
rough = rough * roughFact + 1 - rougFact;

//Converge lighting toward 1 along 'rough'
light /= rough * (1 - light) + light;
```

Enough rimming? I think not, we have a wide one! This one comes from the intuition that we like to light it from behind, this emphasizes that, so that even when we only light it from the side, it still looks kinda backlit. I called it roughness, kinda looked like the oren-nayer rimming effect.

Exponential Tonemapping

```
//Light stored in buffer as exp2(-light)
float3 lightBuf = tex2D(_LightBuffer, uv);

//Invert to get tonemapped lighting info
float3 light = 1 - lightBuf;
```

Thin Depth Offset Rimlight

```
//Offset screenspace uv by a few pixel
//along view norm, sample depth at offset
float2 uvOfs = uv + vNorm.xy * rimWidth;
float zOfs = tex2D(_DepthBuffer, uvOfs);

//Get delta from regular frag depth
float rim = saturate(fragPos.z - zOfs);

//Apply rim to lighting with parameter
light += rim * light * rimFact;
```

Wide Rim aka Roughness

```
//Dot product between view dir and normal
//Then tone it down with parameter
float rough = saturate(dot(view, norm));
rough = rough * roughFact + 1 - rougFact;

//Converge lighting toward 1 along 'rough'
light /= rough * (1 - light) + light;
```

Contrast Enhance aka Toonify

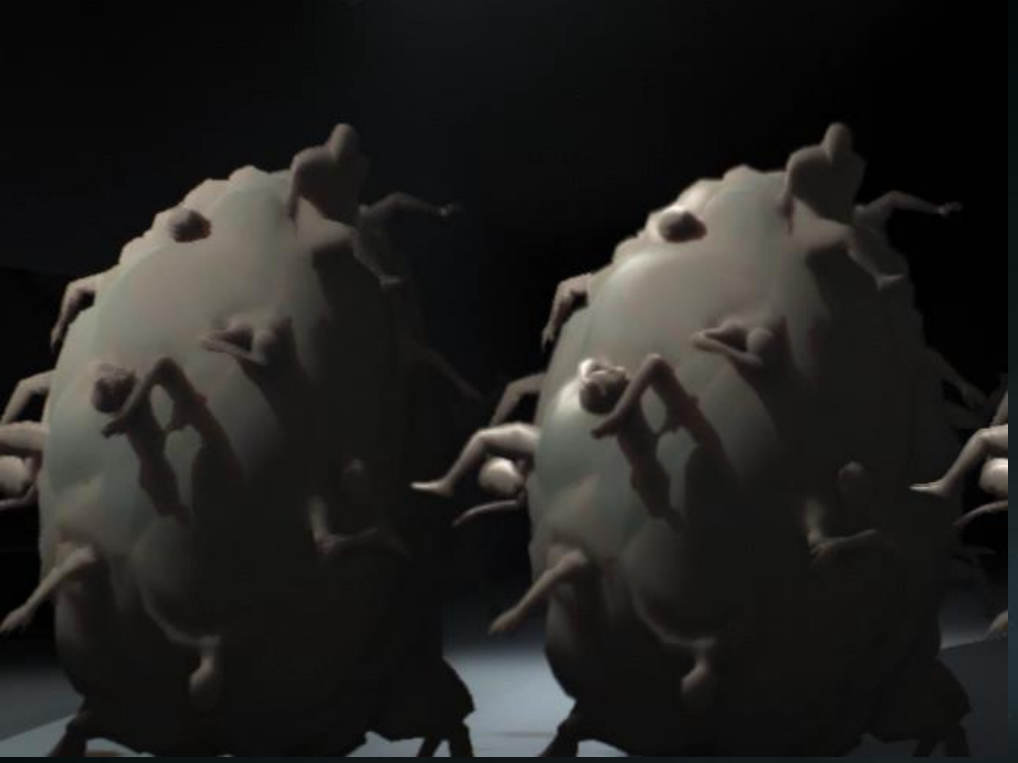
```
//Enhance contrast with smootherstep
//https://en.wikipedia.org/wiki/Smoothstep
float3 toony = smootherstep(0, 1, light);

//Interpolate into it with parameter
light = lerp(light, toony, tooniness);
```

Finally, more contrast! We needed it to fit with the boy, who has a toon ramp, but a toon ramp was too intense for such a large thing, so a very simple smootherstep curve did it. It's basically a smoothstep of a smoothstep. Only special thing here is that we fade this effect on and off depending on how much "in the spotlight" you are, because if it would be on always, that would fuck with the light range falloff.

No Specular

//Nothing to see here...

So that took care of direct lighting, on to some sweat! Without specular, the huddle looks kinda dry, so we needed some spec, here there's obviously none...



No Specular

```
//Nothing to see here...
```

Phong Specular

```
//Phong specular lobe with fake light dir
//Use occlusion as a mask
float3 refl = reflect(lightDir, normal);
float spec = saturate(dot(refl, viewDir));
spec = pow(spec, shininess) * occlusion;

//Use diffuse light as a mask for specular
light += spec * light * (1 - light * 0.5);
```

Here we add a simple phong specular. Nothing special here, except there kinda is, but we'll get to that.

No Specular

```
//Nothing to see here...
```

Phong Specular

```
//Phong specular lobe with fake light dir
//Use occlusion as a mask
float3 refl = reflect(lightDir, normal);
float spec = saturate(dot(refl, viewDir));
spec = pow(spec, shininess) * occlusion;

//Use diffuse light as a mask for specular
light += spec * light * (1 - light * 0.5);
```

Chicken Skin Normal Map

```
//Custom normal for specular only
float3 specNorm = tex2D(specularNormal);

//Use this instead
float3 refl = reflect(lightDir, specNorm);

//Do rest of shading as is...
```

So -normally-, at least with diffuse lighting we reduce normal detail, but here, that ends up looking kinda lube-y, so for just the specular, we add something. This was Marek's idea that it'd be fun if the Huddle looked a bit more like Saturday night chicken.

No Specular

```
//Nothing to see here...
```

Phong Specular

```
//Phong specular lobe with fake light dir
//Use occlusion as a mask
float3 refl = reflect(lightDir, normal);
float spec = saturate(dot(refl, viewDir));
spec = pow(spec, shininess) * occlusion;

//Use diffuse light as a mask for specular
light += spec * light * (1 - light * 0.5);
```

Chicken Skin Normal Map

```
//Custom normal for specular only
float3 specNorm = tex2D(specularNormal);

//Use this instead
float3 refl = reflect(lightDir, specNorm);

//Do rest of shading as is...
```

Fake Arc Light Direction

```
//Light comes from center, top, back
float3 lightDir = float3(0.0, 0.9, 0.8);

//Bend the direction to follow the normal
lightDir.x += softNorm.x * 1.6;
lightDir.z += softNorm.z * 0.5;
```

So, that trick of ours. We don't actually -have- specular in our lighting pipeline, cause we like explicit control over it, as in we have separate diffuse and specular light sources. We didn't wanna add spec light sources to the huddle all over, we wanted it to always look the same kinda sweaty, so instead we calculate a fake specular lobe, and mask it with the diffuse lighting buffer, so it looks like it's specularly lit from above, behind and the sides, a specular rim if you will.

Screen Space Sub Surface Scattering Off

```
float2 pha; sincos(rnd.x * TAU, rot.x, rot.y);
float4 mag = sqrt(rnd.y * 0.25 + float4(0.0, 0.25, 0.5, 0.75)) * _SSSWidth;
//Sample 4 times with different phases of spiral
float4 sss  = blurSample(sPos + pha                   * mag.x, sDepth);
       sss += blurSample(sPos + float2(pha.y, -pha.x) * mag.y, sDepth);
       sss += blurSample(sPos - pha                   * mag.z, sDepth);
       sss += blurSample(sPos - float2(pha.y, -pha.x) * mag.w, sDepth);
//Divide by sample count, blend with lighten to avoid SSS in bright spots
light = max(light, sss.rgb / sss.a * _SSSColor);
```

Finally, on lighting, almost done! We've got some fake subsurface scattering going on. Here it's off...

Screen Space Sub Surface Scattering Off

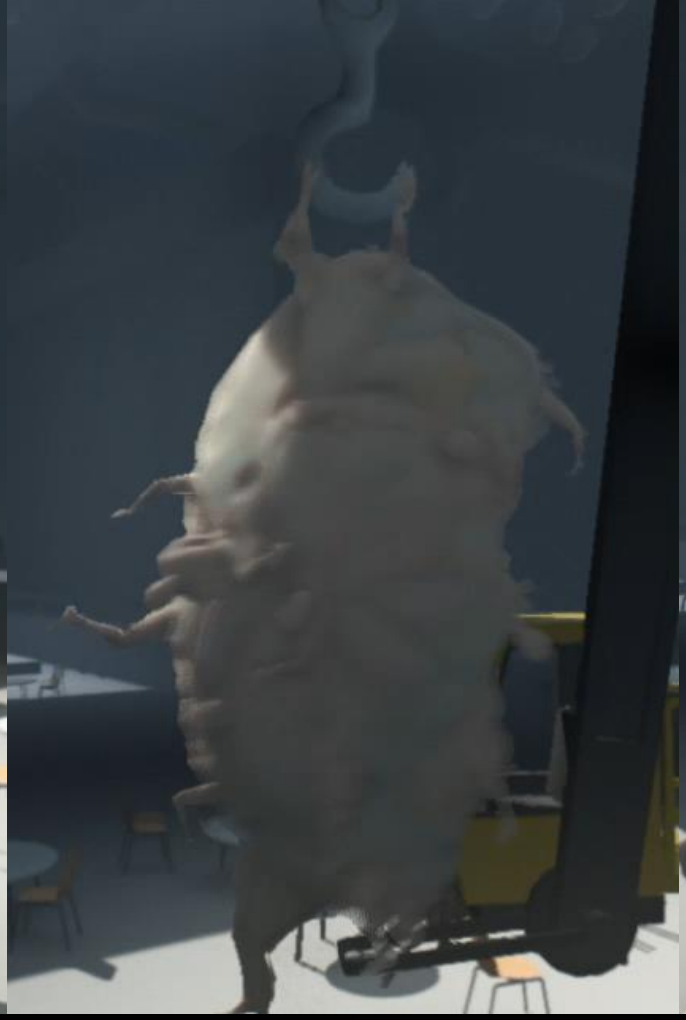
```
float2 pha; sincos(rnd.x * TAU, rot.x, rot.y);
float4 mag = sqrt(rnd.y * 0.25 + float4(0.0, 0.25, 0.5, 0.75)) * _SSSWidth;
//Sample 4 times with different phases of spiral
float4 sss  = blurSample(sPos + pha                   * mag.x, sDepth);
       sss += blurSample(sPos + float2(pha.y, -pha.x) * mag.y, sDepth);
       sss += blurSample(sPos - pha                   * mag.z, sDepth);
       sss += blurSample(sPos - float2(pha.y, -pha.x) * mag.w, sDepth);
//Divide by sample count, blend with lighten to avoid SSS in bright spots
light = max(light, sss.rgb / sss.a * _SSSColor);
```

Screen Space Sub Surface Scattering On

```
float4 blurSample(float2 sPos, float sDepth)
{
       //Sample both buffers
       float4 light = tex2D(lightBuffer, sPos);
       float depth = tex2D(depthBuffer, sPos);
       //Check depth delta, check light buffer for huddle tag
       bool valid = abs(sDepth - depth) < 0.3 && light.a == 1.0 / 3.0;
       return float4(light.rgb, 1.0) * valid;
}
```

And there it's on... It's kind of just a blur of the light buffer, colored red. The blur is done by gathering 4 samples in a random spiral pattern. Each sample is checked for validity, and we divide the sum of valid samples. The success of each sample is determined by depth distance from spiral center, as well as a kind of "Is this The Huddle" stencil bit. The blend we use is lighten, simply cause we only wanted the dark areas to go red, not the already redish skin areas.

Before



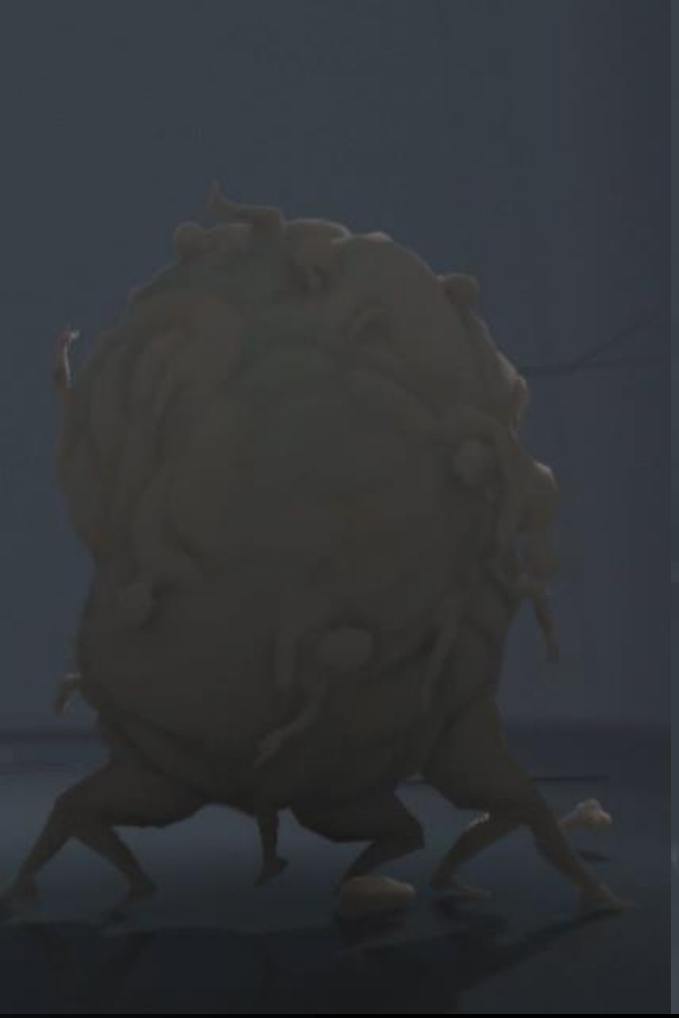
During

After

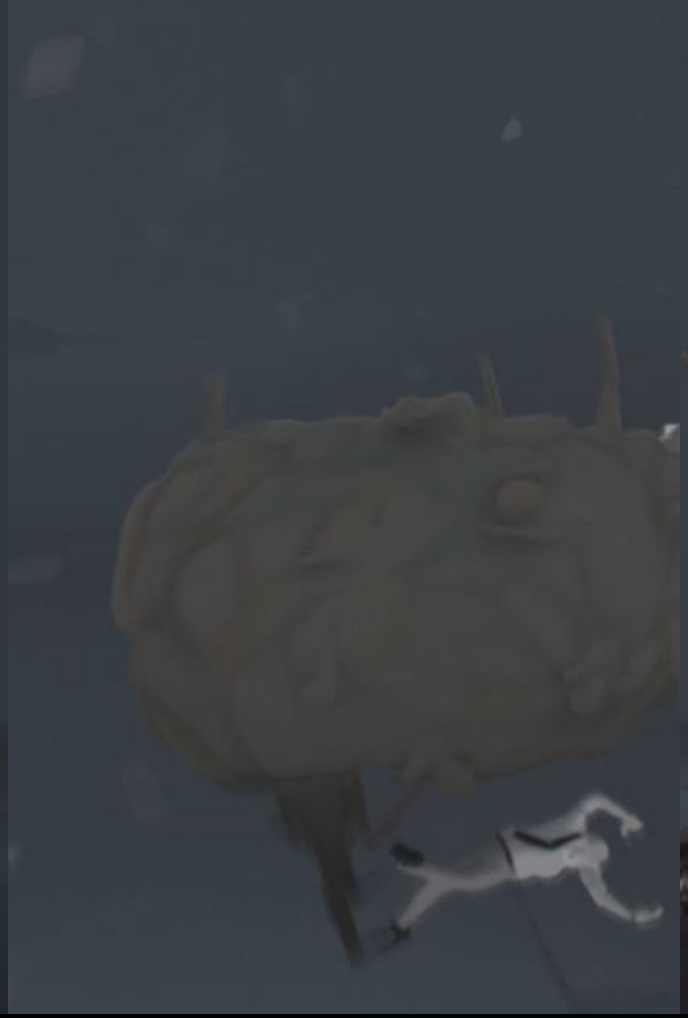
Lastly, as we were wrapping up and feature complete, I thought "Wouldn't it be cool..." and did a thing. The texture of the huddle is a vague mash of color variance, you see subtle blue, green, yellow, red in there, it's not just piggy. This sort of bruise palette would be fun to exaggerate, in some cases. So, here's a non-abused Huddle on the left. And on the right hand side of the right image you can see the these tones exaggerate, and even some bloody scratches. It's mostly on the right side, because, as you can see in the middle, that's where I beat it up. This happens as the audience/player abuses it in a playthrough.

Before During After

Here it's clean again, on the left, and much more bacon-ish on the right. This, of course, because I hung it outside an oven for a while.



Before

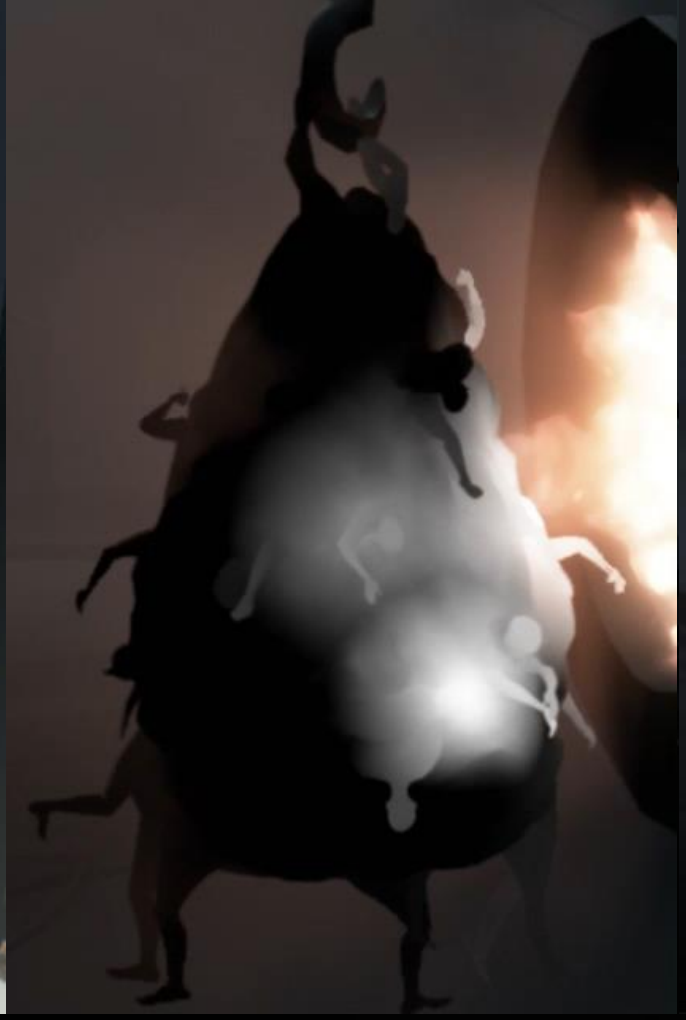


During

After

Finally, here it is again, this time not looking like a murderer, at least on the left, on the right not so much.

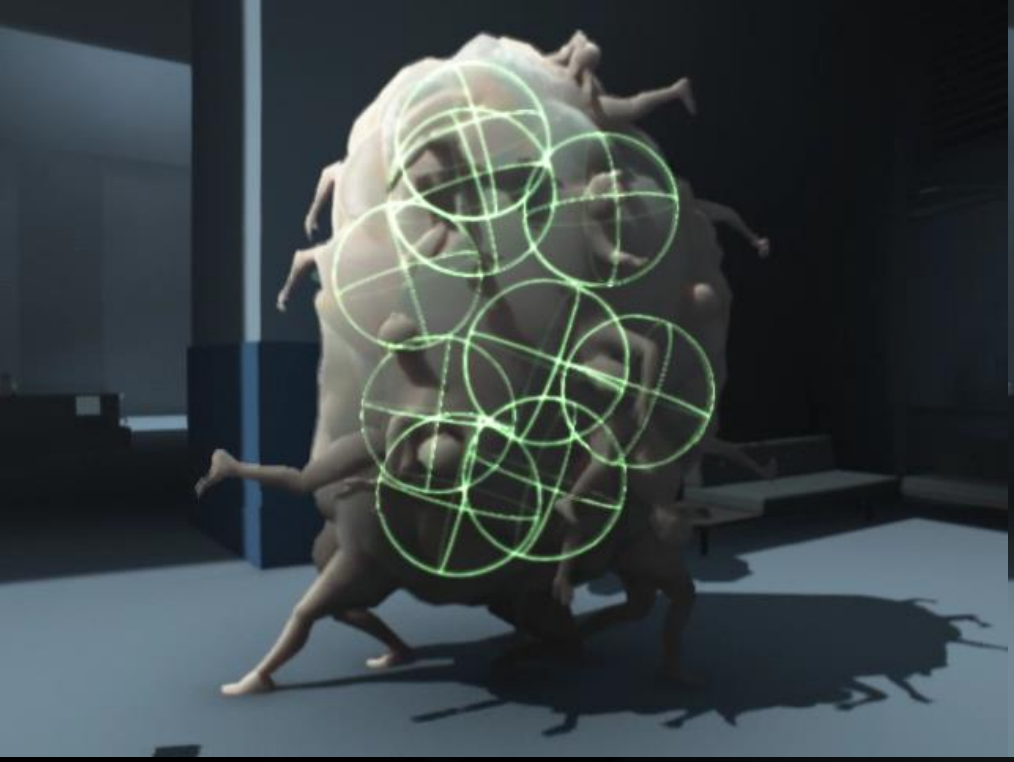
Bruise Mask



Burn Mask

Blood Mask

These things are faded in locally with a mask, the same masking technique for all effects. It's a very low frequency mask, but it's not a render texture.



UV Patches In Script

```
void OnCollisionEnter()
	patchDamage[i].x += collisionSpeed;

void OnTriggerEnter(Collider other)
	if (other == fireTrigger)
		patchDamage[i].y += deltaTime;

public void OnCEOSplat()
	patchDamage[i].z = normalize(center - patchPos[i]).y;
```

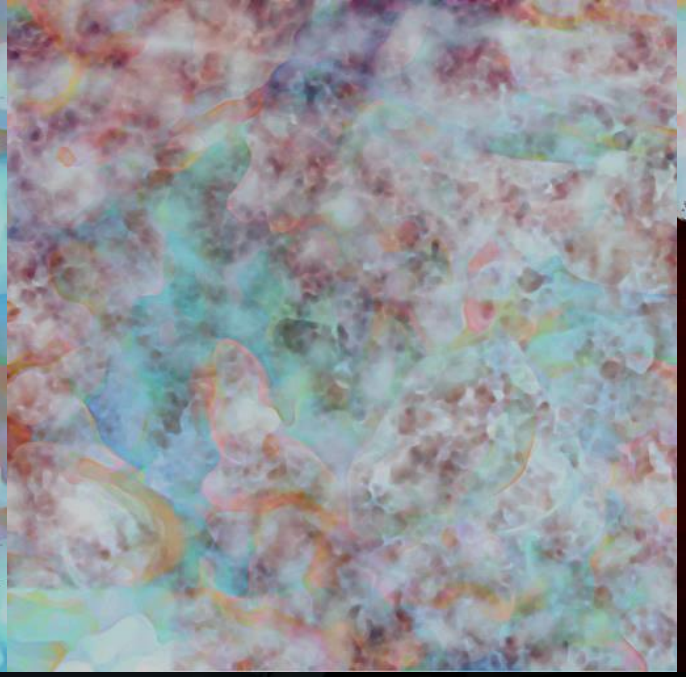
UV Patches In Shader

```
for (uint i = 0; i < 9; i++)
{
	float patchDist = distance(patchPos[i], uv);
	patchDist = invLerp(patchSize[i], 0.0, patchDist);
	damage += patchDist * patchDamage[i];
}
```

The fade is determined by some data from the rigidbodies, my 9 favorite ones out of Lasse's 26, 6 on the rim, 3 in the cetner. Bruises from collision, fire from a trigger, blood from a one time event, that's the CPU work. For the GPU in the shader we have some patches, which I found in UV space and hardcoded, so that in the vertex shader I can create a little mask for some texture effects...

Impact Bruises

```
//Skin tint, always there
float3 skinTint = tex2D(_TintMap, uv);
color *= skinTint;

//Bruisemap has bruises in rgb, bloodmask in a
float4 bruiseCol = tex2D(_BruiseMap, uv);

//damage.x is the bruisemask from vertex program
float bruises = damage.x;

//Blend color with mask
color *= bruiseCol.rgb * bruises + 1.0 - bruises;
```



Burn Wounds

```
//Burnmap has burns in rgb, burnmask in a
float4 burnCol = tex2D(_BurnMap, uv);

//Subtract burnmap details, remap
float burns = damage.y - burnCol.a;
burns = smoothstep(0.0, 0.5, burns);

//Blend color with mask
color *= burnCol.rgb * burns + 1.0 - burns;
```

Blood Splatters

```
//Subtract bruisemap's bloodmask details, remap
float blood = damage.z - bruiseCol.a;
blood = smoothstep(0.0, 0.25, blood);

//Blend color with mask
color *= bloodCol * blood + 1.0 - blood;
```

The different effects are simply making the existing huddle texture more damaged, the impact bruises are just using tint or multiply, the burn wounds the same, except dithered a bit, and the blood splatter as you can see has no colour, just red, but with a very hard fade, and also it prefares to add blood on the rim rather than the center, where it would rather put a bit of splatter.

Aaand I think we've got enough into shading detail now! It was a lot fun making The Huddle, but I didn't make it, these guys did. I wish I knew how to wrap this up, but...

Thank you!
Andreas Normand Grøntved https://twitter.com/andreasng____
Lasse Jon Fulgsang Pedersen https://twitter.com/codeverses
Søren Trautner Madsen https://twitter.com/sorentrautner
Mikkel Bøgeskov Svendsen https://twitter.com/IkarosAV