Class 7

User Interfaces and Skeletal Animation

User Interfaces

INTRODUCTION

Many shades of UI

- Heads-up display (HUD)
  - Persistent 2D elements drawn over the game
    - Ex. your health & resources, minimap, available actions/cooldowns
- Orthographic UI
  - 2D elements drawn in the 3D scene
    - Ex. entity health bars, tips, popup dialog, 2D special effects
    - Usually doesn't scale with camera distance
- Separate screens/overlays
  - Inventory, menus, shops, dialog

Heads-Up Display

Game stats (KDA, fps)

Player health & abilities (most important)

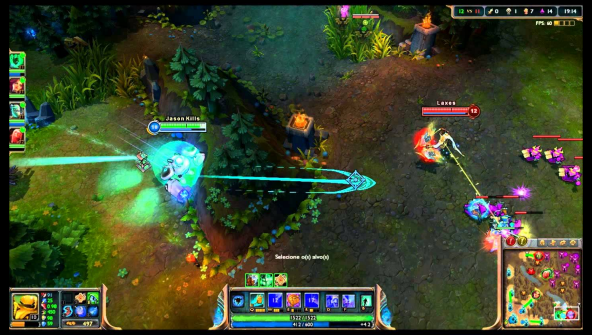
Minimap

HUD

(League of Legends)

Player stats & inventory

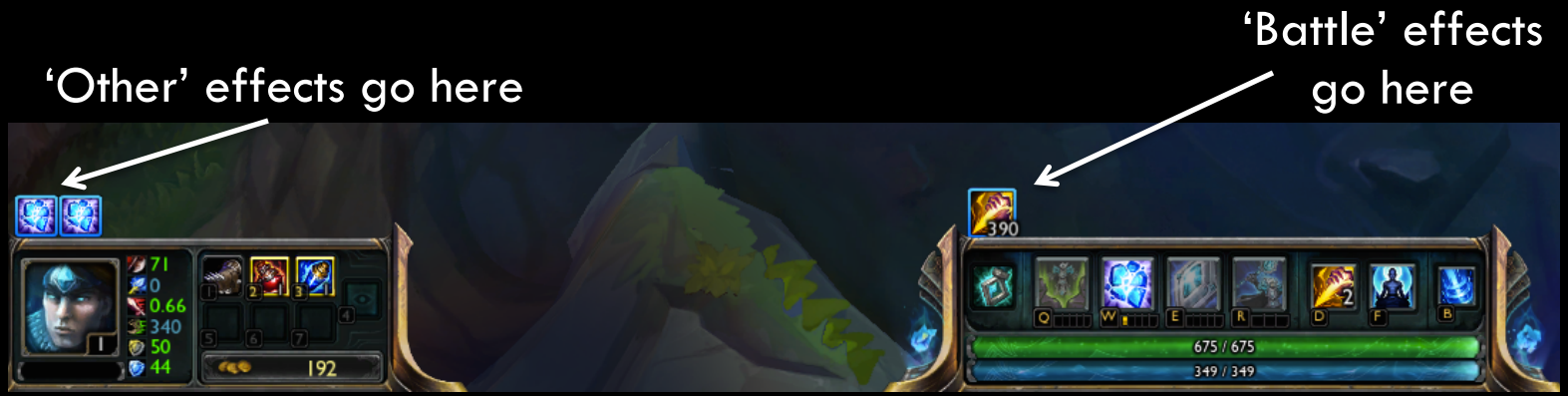
HUD Clarity



- The HUD takes up minimal screen space
- Each different type of information is placed in a distinct and isolated area of the screen

HUD Clarity

- Deciding where to put information (and how to group it) is very important



Drawing HUD

- Use the built in `m_camera->setUI(true)` so that your draw calls act in screen space
- Draw things over your game!

User Interfaces

ORTHOGRAPHIC

Orthographic UI

Tooltips

Health bars & Damage

Orthographic UI

“MEDIC!”

(Team Fortress 2)

Dialog

(Runescape)

How to Orthographic UI

- Goal: draw orthographic UI next to / above an entity within your game
- Approach
  1. Figure out screen space position of the object you want to draw
  2. Figure out if the game object is in front of or behind the camera
  3. Draw orthographic UI at that location (same way to draw HUD)

Converting to screen space

```
vec3 convertToScreenSpace(vec3 pos, vec2 screenSize) {
    // get these from your camera
    mat4x4 view, projection;
    // this is necessary for the matrix multiplication
    vec4 fourVec = vec4(pos.x, pos.y, pos.z, 1.f);
    // project the position into clip space
    fourVec = projection * view * fourVec;
    // x and y range from -1 to 1
    vec3 clipSpace = vec3(fourVec.x / fourVec.w, fourVec.y / fourVec.w,
                          fourVec.z);
    // convert x and y to pixel coordinates, leave z alone
    return vec3((clipSpace.x + 1) * .5f * screenSize.x,
                (1 - clipSpace.y) * .5f * screenSize.y, clipSpace.z);
}
```

Using the projected position

- Use the x and y components of the vector calculated in the previous slide to position draw calls on the screen
- If the z-component is less than 0, it means that the object is behind the camera
  - In other words, don't draw it!
- If it's greater than 0, draw the UI!
- If you want your orthographic UI elements to be occluded by objects in the world, make sure the OpenGL depth test is turned on
  - **When you use a UI camera, the stencil turns off the depth test by default**

Users Interfaces

ENGINE INTEGRATION

How to integrate UI?

Recommended:

- `GameWorld` has a `UISystem` with `UIComponents`.
- Tell your `UISystem` to draw after drawing all the other components
- If an orthographic UI component requires some game logic, have it belong to (and have a reference to) a `GameObject`.

Your UI

- You are required to implement some use of orthographic UI and HUDs this week
- Try to think of something unique and interesting to your game concept!
    - E.g. inventory, health bars

More UI

- Some UI is related to `GameWorld`
  - In game pop-ups, shopkeeper menus, etc.
- Some UI isn't
  - Settings menu, pause menu, etc.
- It's okay to handle these differently

3D-Integrated UI

Instructions

(Splinter Cell)

Minimap

(Far Cry 2)

3D-Integrated UI

All user interfaces as in-game holograms

(Dead Space)

User Interfaces

QUESTIONS?


Class 7
Skeletal Animation

SKELETAL ANIMATION (FROM A HIGH LEVEL)

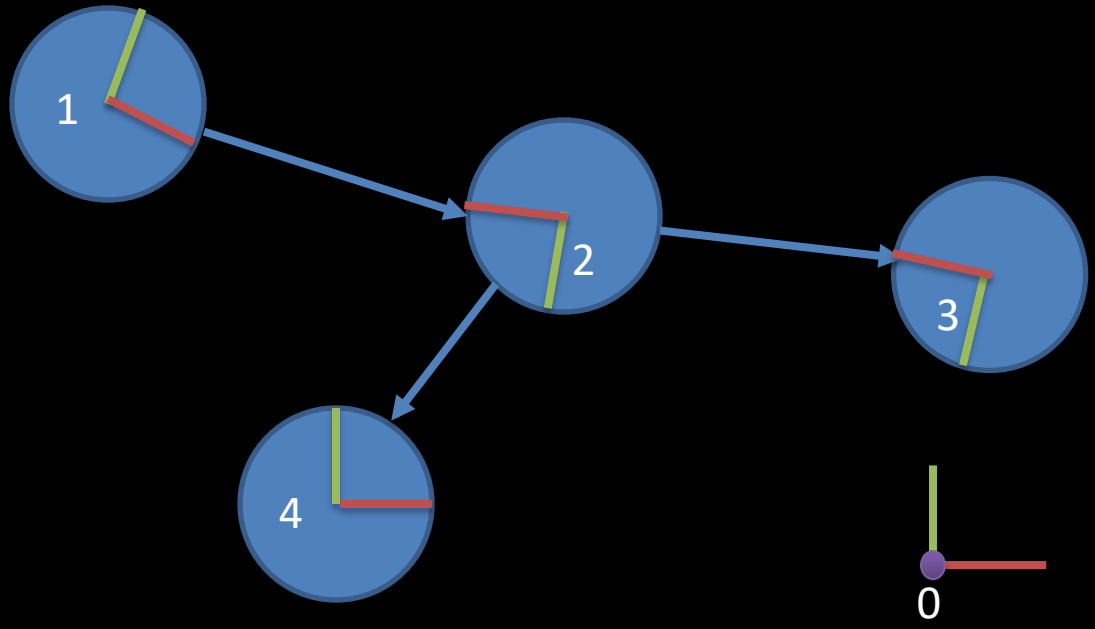
Skeletal Animation

- How can we animate meshes in our game engines?
- We will need some method of moving vertices at each tick
- But storing the position of each vertex in the mesh for each frame of the animation would be expensive

Skeletal Animation

- The method of skeletal animation uses a “skeleton” of joints that control the positions of the vertices (i.e. ”the skin”)
- We can store the pose of the skeleton at each frame, and then use the skeleton to compute the vertex positions

The Skeleton



The Skeleton

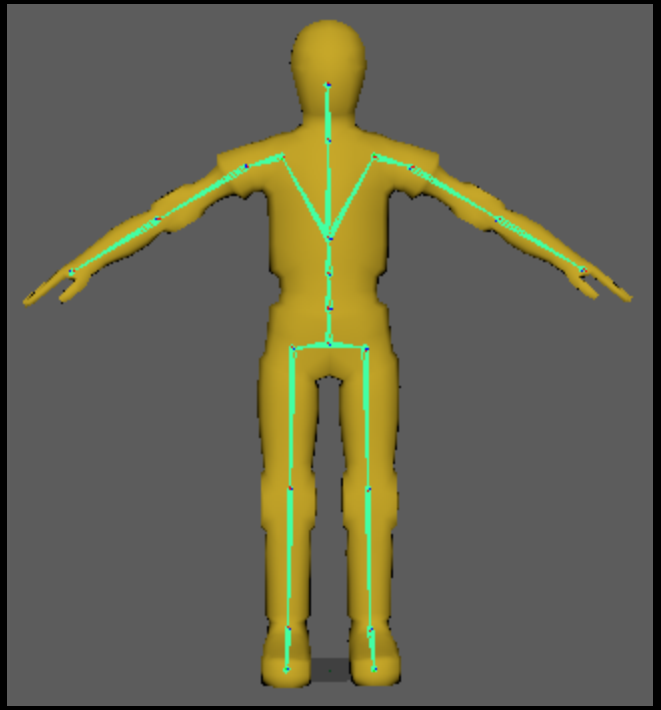
- A skeleton is a **hierarchy** of nodes called “joints”
- A joint has a position and an orientation that can change at each tick
- Here, the red (x) and green (y) axes represent each joint’s orientation (i.e. the local coordinate system of each joint)
- More on skeletons later…

Joint Weights

- But how does the skeleton control the mesh?
- Each joint has a certain constant influence on each vertex, usually called a "weight"
  - If the weight of joint *j* on vertex *v* is 1.0, then *j* completely controls *v*
  - If the weight is 0, then *j* has no influence on *v*
  - If the weight is between 0 and 1.0, then *v* is influenced by joint *j* and by other joints as well

Skinning

- We define the joint weights in a process called “skinning”
- First, we take the mesh in the “bind pose” (for a character, this is usually a T-pose)
- Next, we take the skeleton (also in the bind pose)
- Let the joint weight of joint j for vertex v be w_{ij}
- We can use many different methods to calculate the joint weights, but in general, if vertex v is far away from joint j, then w_{ij} is small or 0
- If vertex v is close to joint j, then w_{ij} will be close to 1.0



Animation

- Now we have a mesh bound to a skeleton, and we know the influence that each joint has on each vertex of the mesh
- We can think of an ”animation” as a sequence of poses of the skeleton
- **Using the skeleton’s bind pose, the skeleton’s animated pose, and the joint weights, we can calculate the position of each vertex for each frame of the animation**

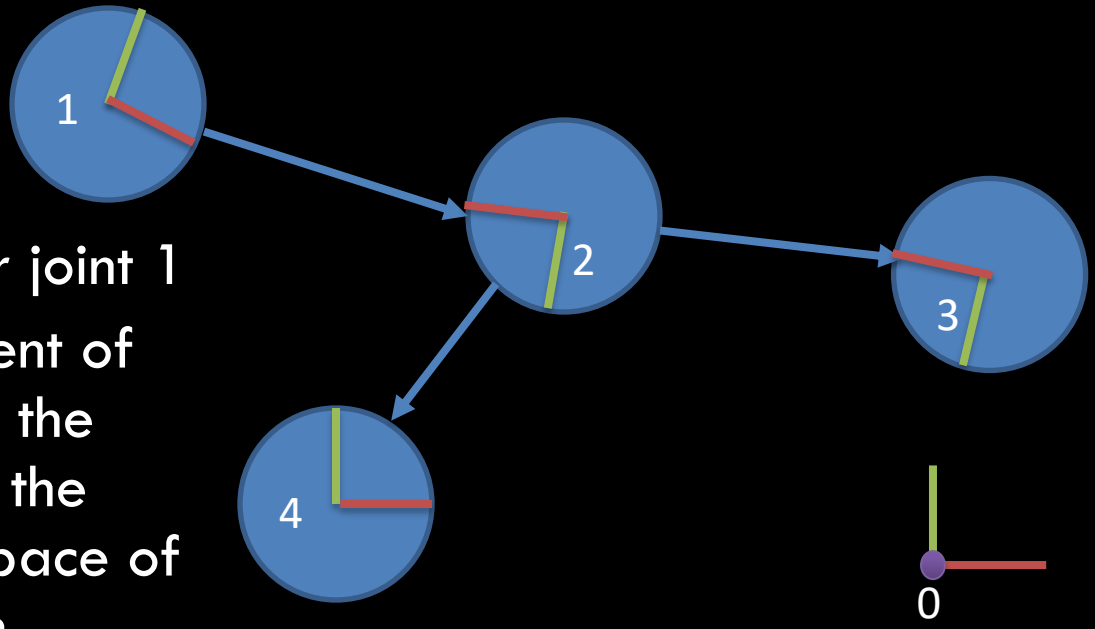
THE SKELETON

Defining the Skeleton

- Remember that a skeleton is a hierarchy of joints
  - Each joint can have an arbitrary number of children, and each joint has a parent joint
- In this class, we will define the position and orientation of each joint *j* using a **rotation R and translation T in the coordinate system of *j*'s parent**
  - **We will call these types of transforms "joint-space transforms"**
- What does that mean?

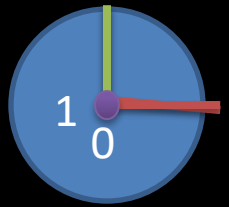
Defining the Skeleton

- Consider joint 1
- The parent of joint 1 is the origin in the object space of the mesh



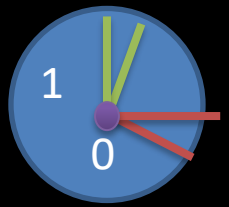
Defining the Skeleton

- Starting at the origin…

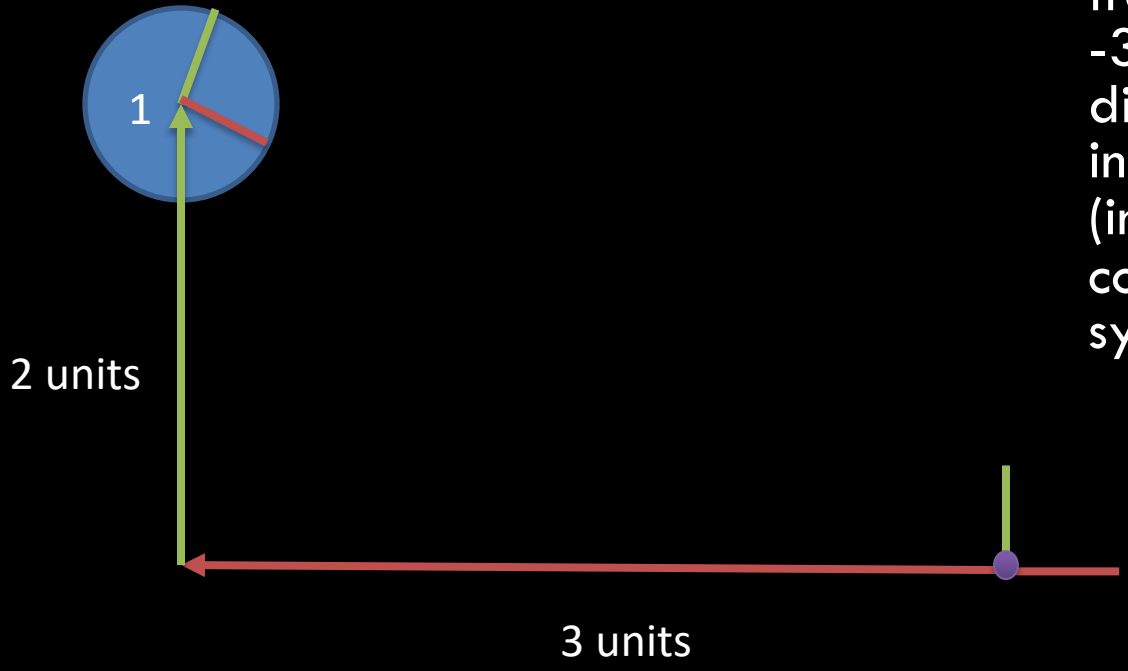


Defining the Skeleton

- We can rotate joint 1 about 20 degrees clockwise (in the parent's coordinate system)…



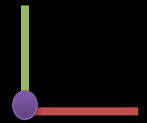
Defining the Skeleton



- And then translate the joint -3 units in the x direction +2 units in the y direction (in the parent's coordinate system)

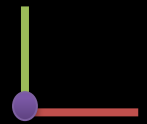
Defining the Skeleton

- Let's look at another example



Defining the Skeleton

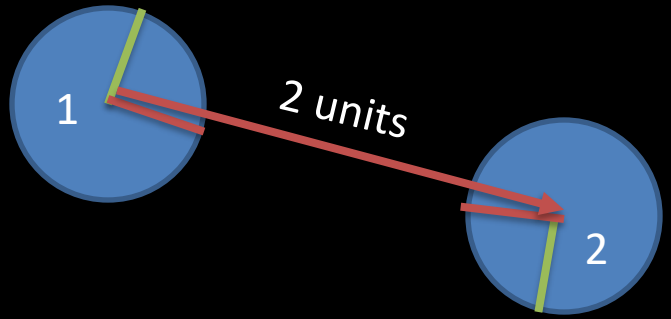
- Joint 2 starts at its parent (with the same position and orientation)



Defining the Skeleton

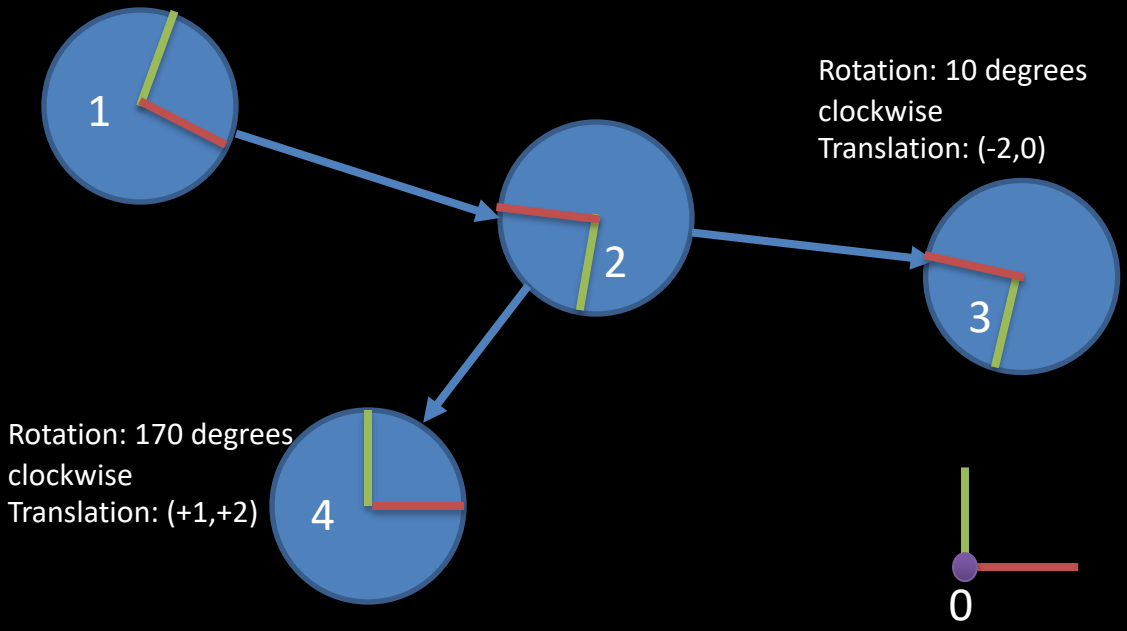
- Then we rotate joint 2 170 degrees clockwise (in the parent's coordinate system)

Defining the Skeleton



- Then we translate +2 units in the x direction (in the parent's coordinate system)

Defining the Skeleton



- Here are the joint-space transforms for joints 3 and 4

Defining the Skeleton

- The skeleton's bind pose and its poses in an animation are defined using joint-space transforms
- The joint space transform is represented as a 4x4 matrix (joint_space_transform = translation * rotation)

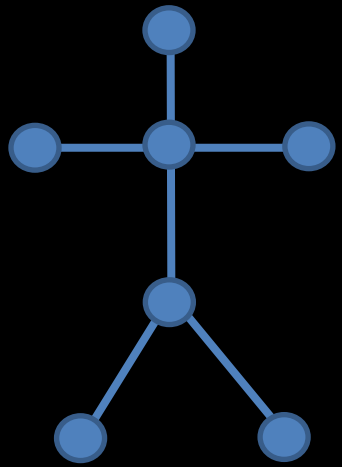
CALCULATING ANIMATED VERTICES

Animating Vertices

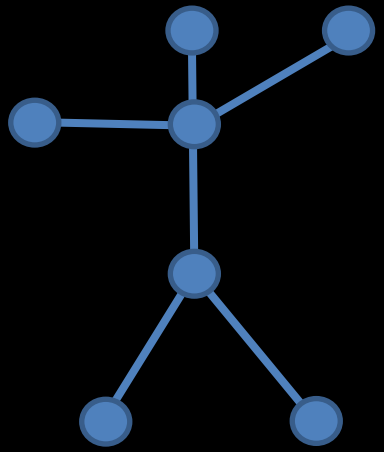
- Given the joint space transforms of a skeleton's bind pose p_b, the joint space transforms of an animation pose p_a, and the vertex positions of the mesh in p_b, we can calculate the vertex positions of the mesh in p_a
- In the following slides, we will make a simplifying assumption: **Each vertex is influenced by exactly one joint**
  - We will get rid of this simplification soon

Animating Vertices

- If each vertex is influenced by exactly one joint, then the position of vertex *v* relative to its influencing joint *j* should be constant, no matter what pose the skeleton is in
  - Think of a point on your forearm
  - It is influenced completely by the position and orientation of your elbow, and it stays at a constant position relative to your elbow's position and orientation



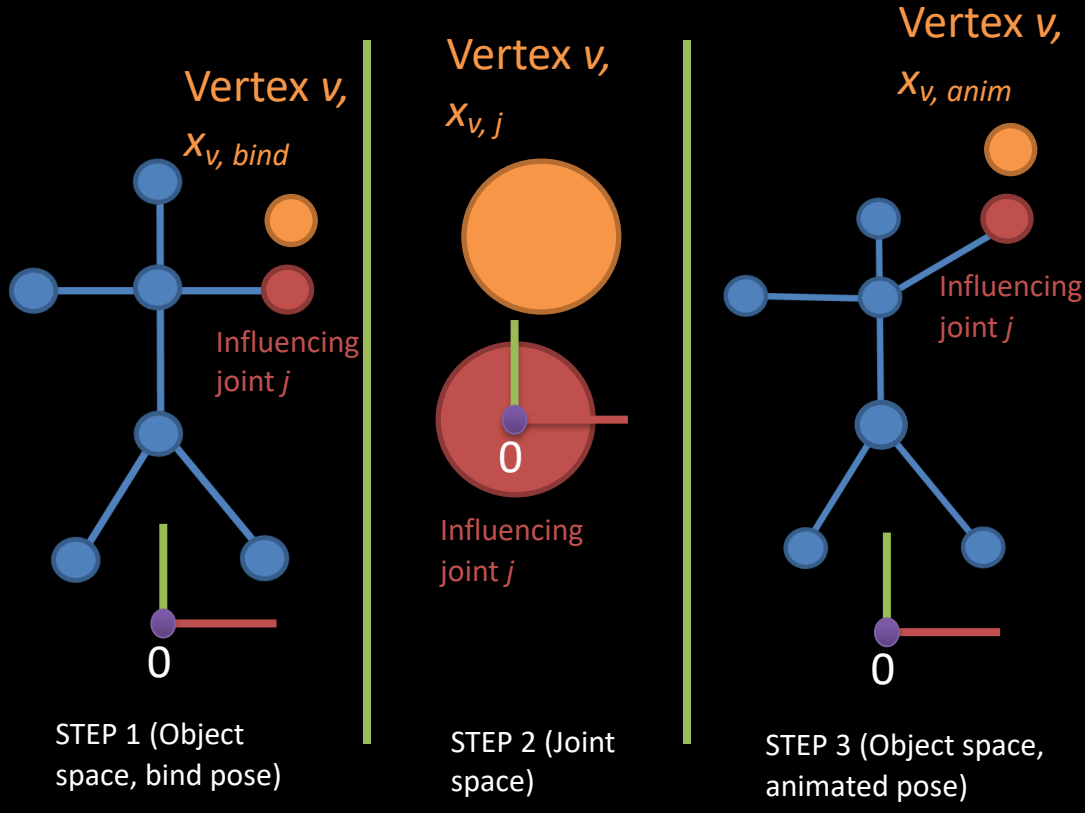
Bind Pose



Animation Pose

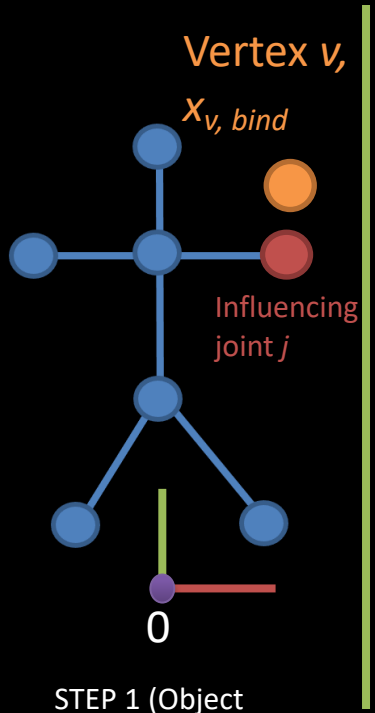
Animating Vertices

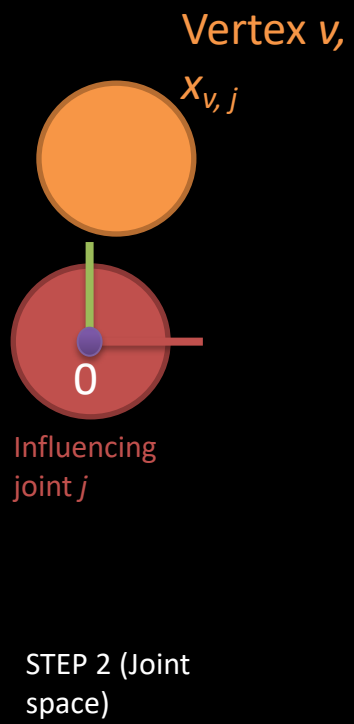
- We are given the coordinates of vertex v in the bind pose (call them $x_{v,\,bind}$)
  - We will first calculate the coordinates of v in the joint-space of the influencing joint j (Call these coordinates $x_{v,j}$)
  - From the previous slide, we know $x_{v,j}$ will not change when the pose changes
- Next, we will convert $x_{v,j}$ to coordinates in object space, when joint j is in pose p_a (call them $x_{v,\,anim}$)



Animating Vertices

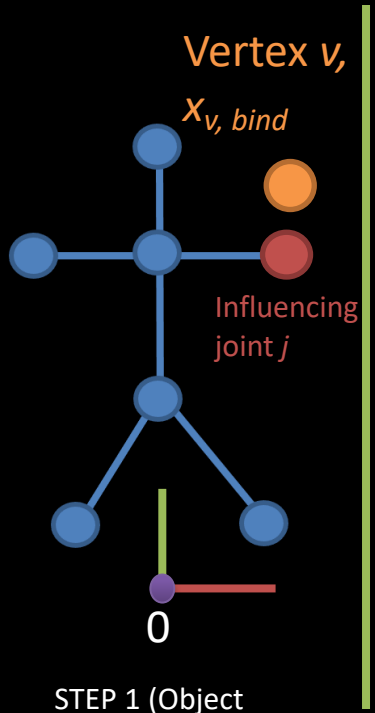
- We are given step 1, so how do we get to step 2?
- Let $P(j)$ be the parent of joint j
- Let S_j be the joint-space bind pose transformation matrix of joint j
- Define $T_{root} = S_{root}$, the object space bind pose transformation matrix of the root joint in the skeleton
- Define $T_j = T_{P(j)}S_j$, the object space bind pose transformation matrix of joint j

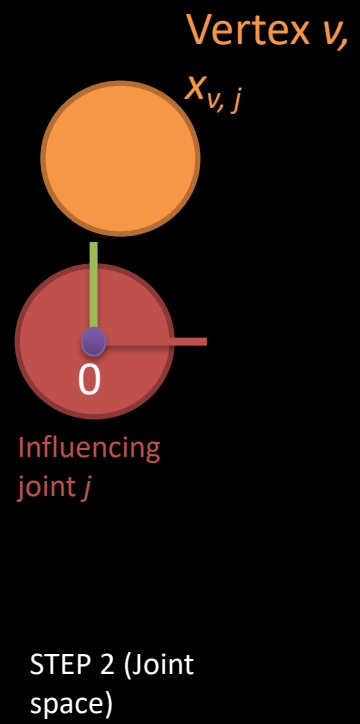




Animating Vertices

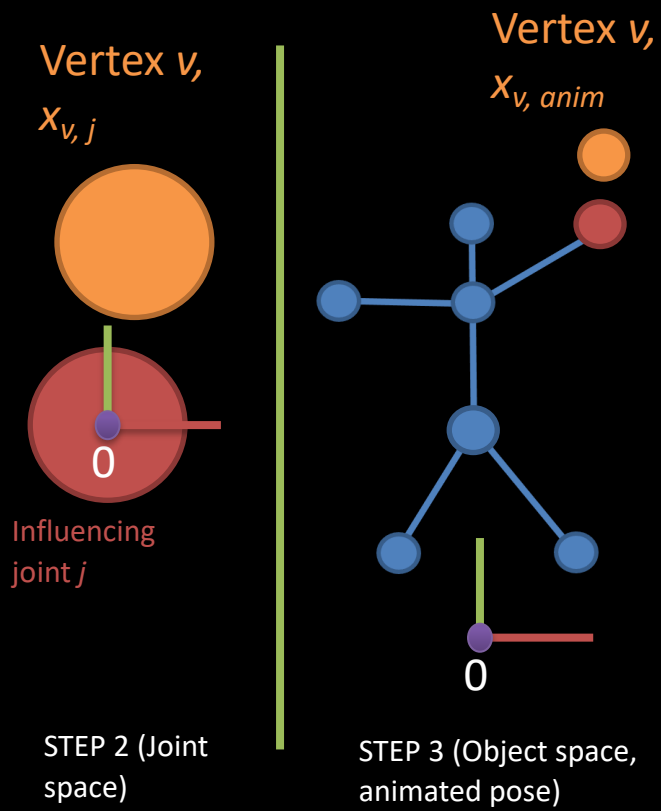
- Define $T_j = T_{P(j)}S_j$, the object space bind pose transformation matrix of joint j
- Using the fact that S_j is a rotation followed by a translation (slide 21), we can show that **T_j takes a point in the joint-space of joint j and outputs the same point in object space with the skeleton is in the bind pose**
- So, we can take the inverse of T_j, which will take a point in object space with the skeleton in the bind pose and output the same point in the joint-space of joint j
  - inverse(T_j) is called the "inverse bind transform" for joint j
- $X_{v,\,j}$ = inverse(T_j) * $x_{v,\,bind}$
- So, step 2 is complete!





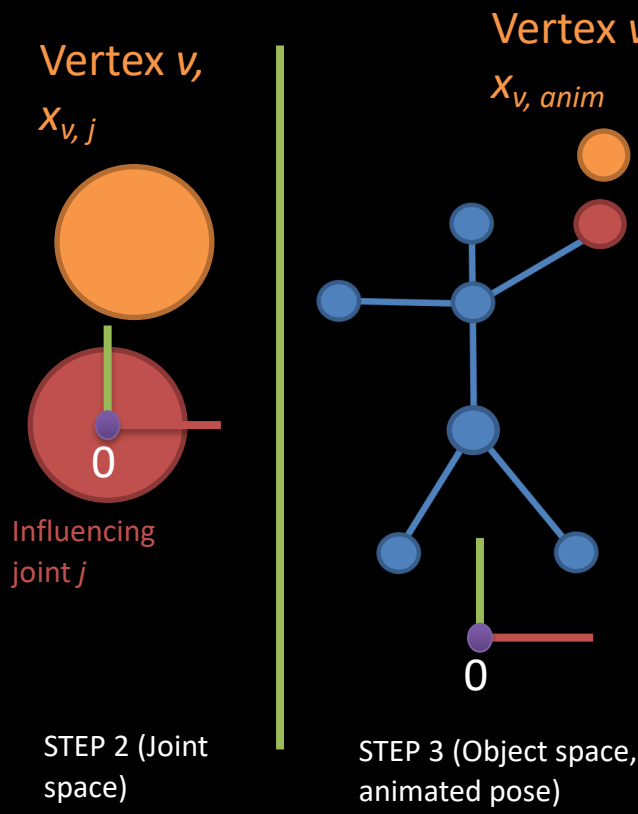
Animating Vertices

- How do we go from step 2 to step 3?
- Remember we are just dealing with a single frame of animation here
- Let P_j be the joint-space **animated** pose transformation matrix of joint j
- Define $A_{root} = P_{root}$, the object space **animated** pose transformation matrix of the root joint in the skeleton
- Define $A_j = A_{P(j)}P_j$, the object space **animated** pose transformation matrix of joint j



Animating Vertices

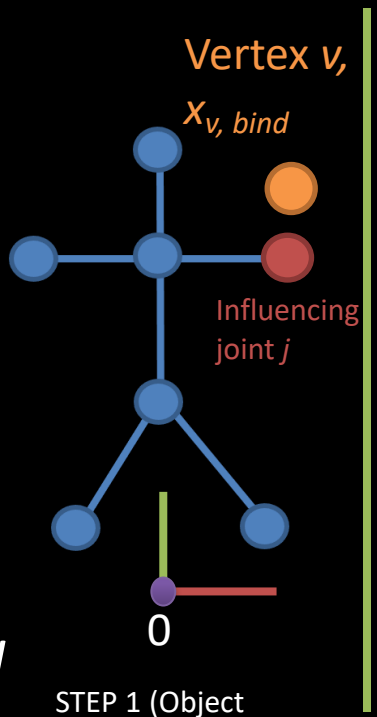
- Define $A_j = A_{P(j)}P_j$, the object space **animated** pose transformation matrix of joint *j*
- Just as T_j takes a point in the joint-space of joint *j* and outputs the same point in object space with the skeleton **in the bind pose…**
- A_j takes a point in the joint-space of joint *j* and outputs the same point in object space with the skeleton **in the animated pose**



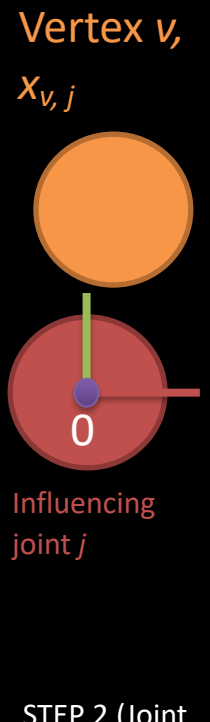
Animating Vertices

- Now we know how to animate vertices!
- Take the position of vertex in the bind pose: $x_{v,\ bind}$
  - Remember v is ony influence by joint j
- Then, the vertex position in the animated pose is…

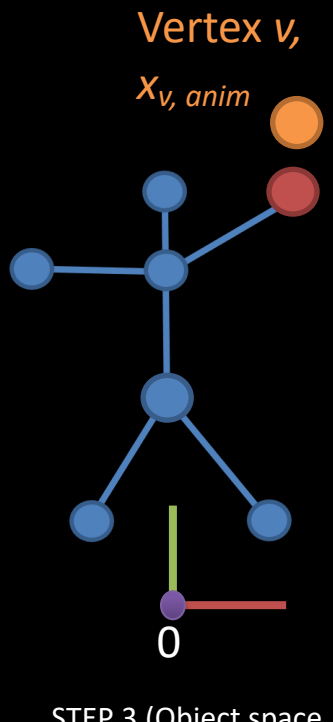
$$x_{v,\ anim} = A_j T_j x_{v,\ bind}$$



STEP 1 (Object space, bind pose)

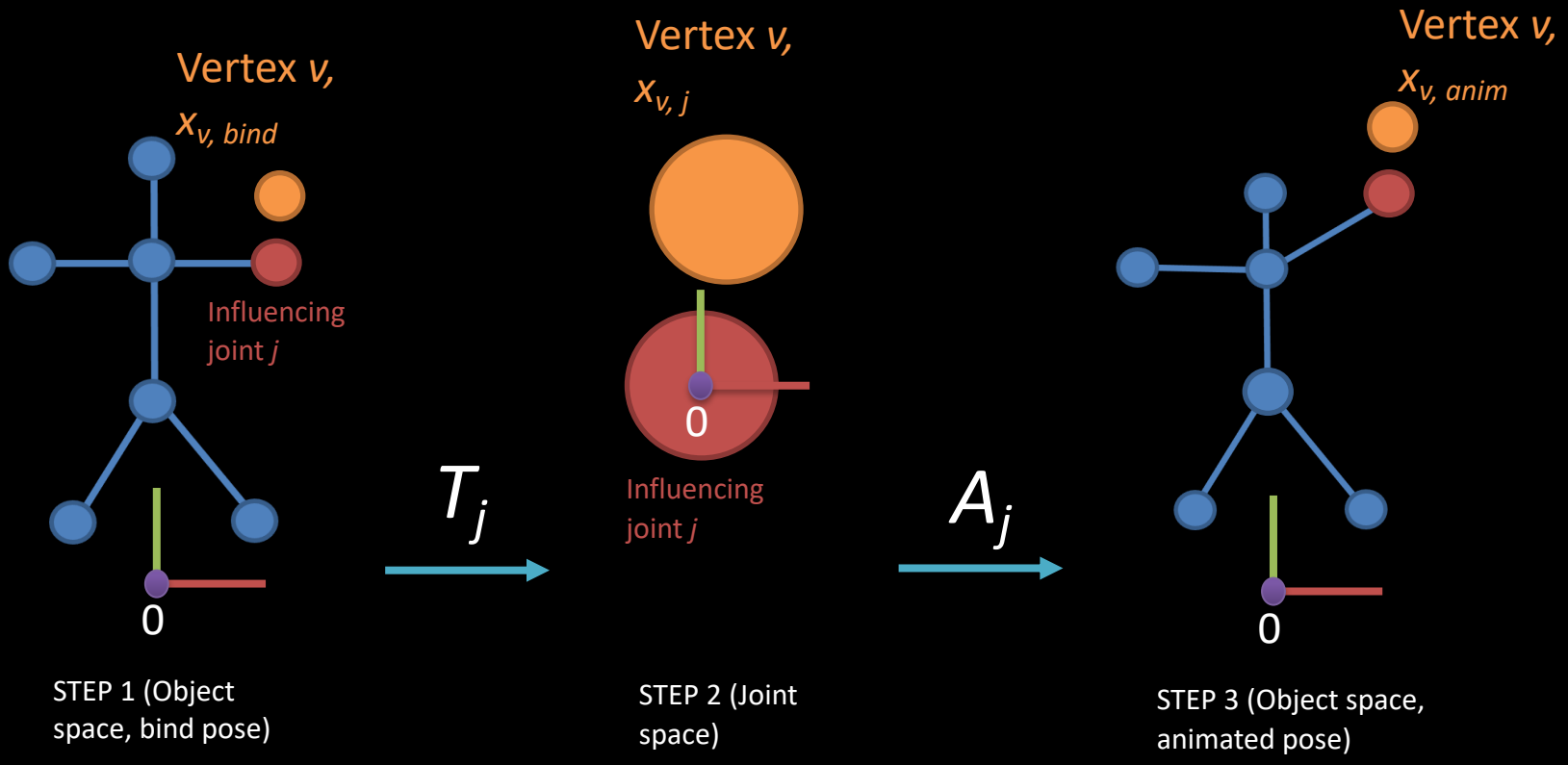


STEP 2 (Joint space)



STEP 3 (Object space, animated pose)

Animating Vertices



STEP 1 (Object space, bind pose)

STEP 2 (Joint space)

STEP 3 (Object space, animated pose)

Animating Vertices

- But how do we deal with vertices influenced by multiple joints?
- Let $j_1,\ldots,j_n$ be the influencing joints on joint v, and let $w_1,\ldots,w_n$ be the corresponding weights
- Then we just take a weighted average

$$x_{v,\,anim} = \sum w_i A_i T_i x_{v,\,bind}$$

EXTRA DETAILS

Interpolating Frames

- A tick will not necessarily take place right at the instant a frame of the animation is supposed to take place!
- We can save space by defining fewer frames in an animation
- What do we do if the tick doesn't occur at the instant of the frame?
- **We need to *interpolate* two frames**

Interpolating Frames

- If the current tick occurs between frames f_0 and f_1 (at a time between t_0 and t_1), then we need to produce interpolated frame data
- f_{interp} = interpolate(f_0, f_1, $t_0/(t_1 - t_0)$)
- *To get* f_{interp}, we need to interpolate the rotation and translation information of f_0 and f_1

Interpolating Translation Data

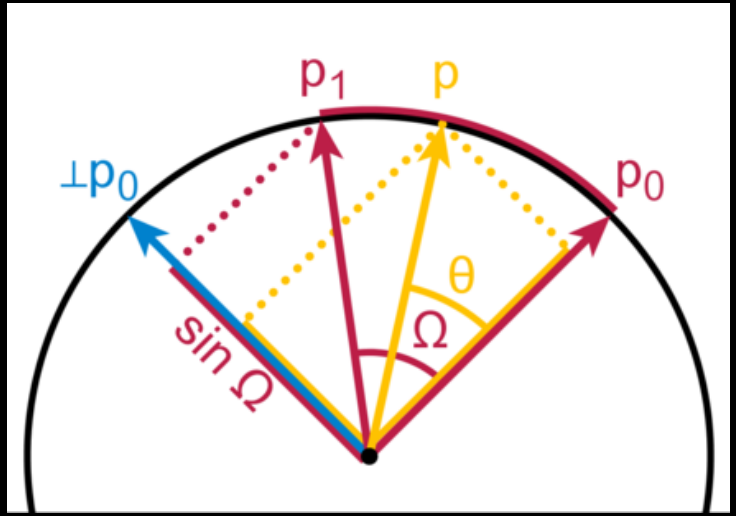
- To linearly interpolate the translation data, all we need to do is take $x_{interp} = x_0 + (x_1 - x_0) * t_0/(t_1 - t_0)$, where x_0 is the translation data from frame f_0 and x_1 is the translation data from frame f_1

Interpolating Rotation Data

- To linearly interpolate the rotation data, we can't just perform traditional interpolation on the rotation matrices
- We need to represent the rotation data of the frames as **unit quaternions**
  - Quaternions are 4-element vectors that represent a rotation about an axis
  - Rotation by θ about $N = (x, y, z)$ gives a quaternion of $(\cos(\theta/2), x \sin(\theta/2), y \sin(\theta/2), z \sin(\theta/2))$
- We perform spherical linear interpolation (SLERP) on the quaternion representation

SLERP

- The SLERP algorithm interpolates the quaternions over the 4D hypersphere
  - If we linearly interpolated the quaternions as we do with translations, we would not necessarily end up with a unit quaternion
- You are free to use glm::mix, which works on glm::quat



Animation Files

- There are many different ways to store animation files
  - COLLADA (.dae)
  - FBX (.fbx)
  - and more

WARNING!

- Implementing skeletal animation can be an enormous time sink because the files that store the animation are very complicated and can work in unintuitive ways!
- It can also be difficult to export multiple animations for the same model in one file
- Using a parsing library will save you some time, but you will probably still have to learn the details of the file type you are using

WARNING!

- Implementing skeletal animation can be an enormous time sink because the files that store the animation are very complicated and can work in unintuitive ways!
- It can also be difficult to export multiple animations for the same model in one file
- Using a parsing library will save you some time, but you will probably still have to learn the details of the file type you are using

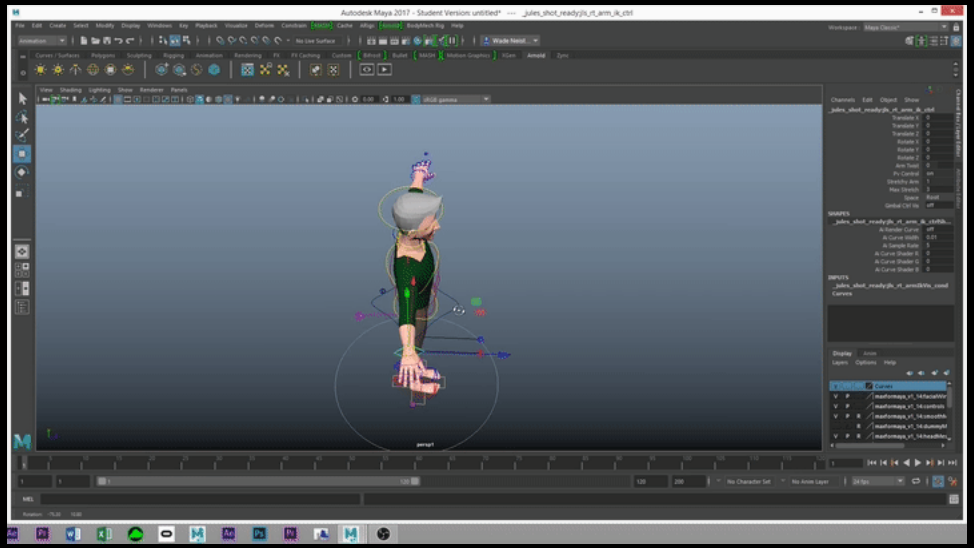
WARNING!

- If you want to implement skeletal animation, you should probably use COLLADA files because they are easier to understand compared to FBX files

ADVANCED ANIMATION

Inverse Kinematics

- “Forward kinematics” is the process of positioning the joints of a skeleton from the root to the leaves of the hierarchy
- “Inverse kinematics” is the process of positioning the end of a limb (like a foot or a hand) and automatically solving for the positions of the joints that result in the



Blend Shapes

- Simpler than skeletal animation!
- We have a start and end position for each vertex, and we just interpolate between those two positions
- Useful for animating faces

Dual Quaternion Blending

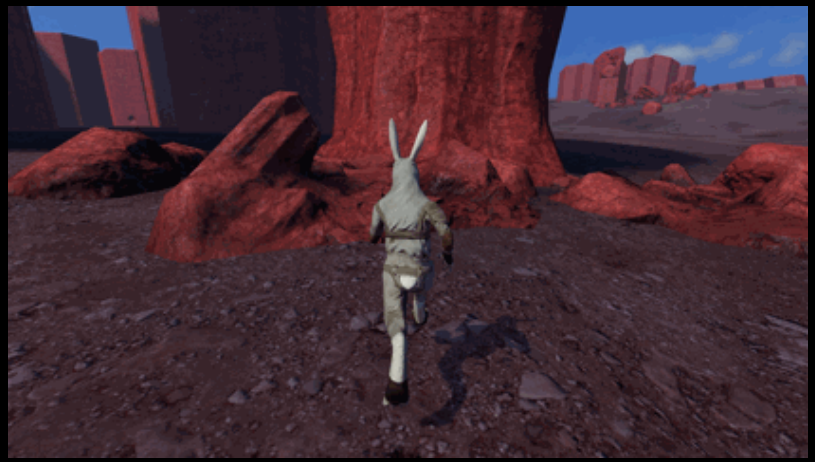
- We have discussed “linear blend skinning”
  - Unfortunately, this type of skinning can result in non-rigid transformations
  - These occur because we add up rigid transformations in our vertex shader, but rotation matrices are not closed under addition!
- “Dual quaternion skinning” takes the same weights, rotations, and translations we use in linear blend skinning and produces final vertex transformations that are rigid
- Fortunately, upgrading from linear blend skinning to dual quaternion skinning is easy!
  - Just convert your matrices to dual quaternions before sending them to the vertex shader, and then modify your vertex shader to properly blend the dual quaternions
  - Check out “Skinning with Dual Quaternions” by Kavan et al. for details!

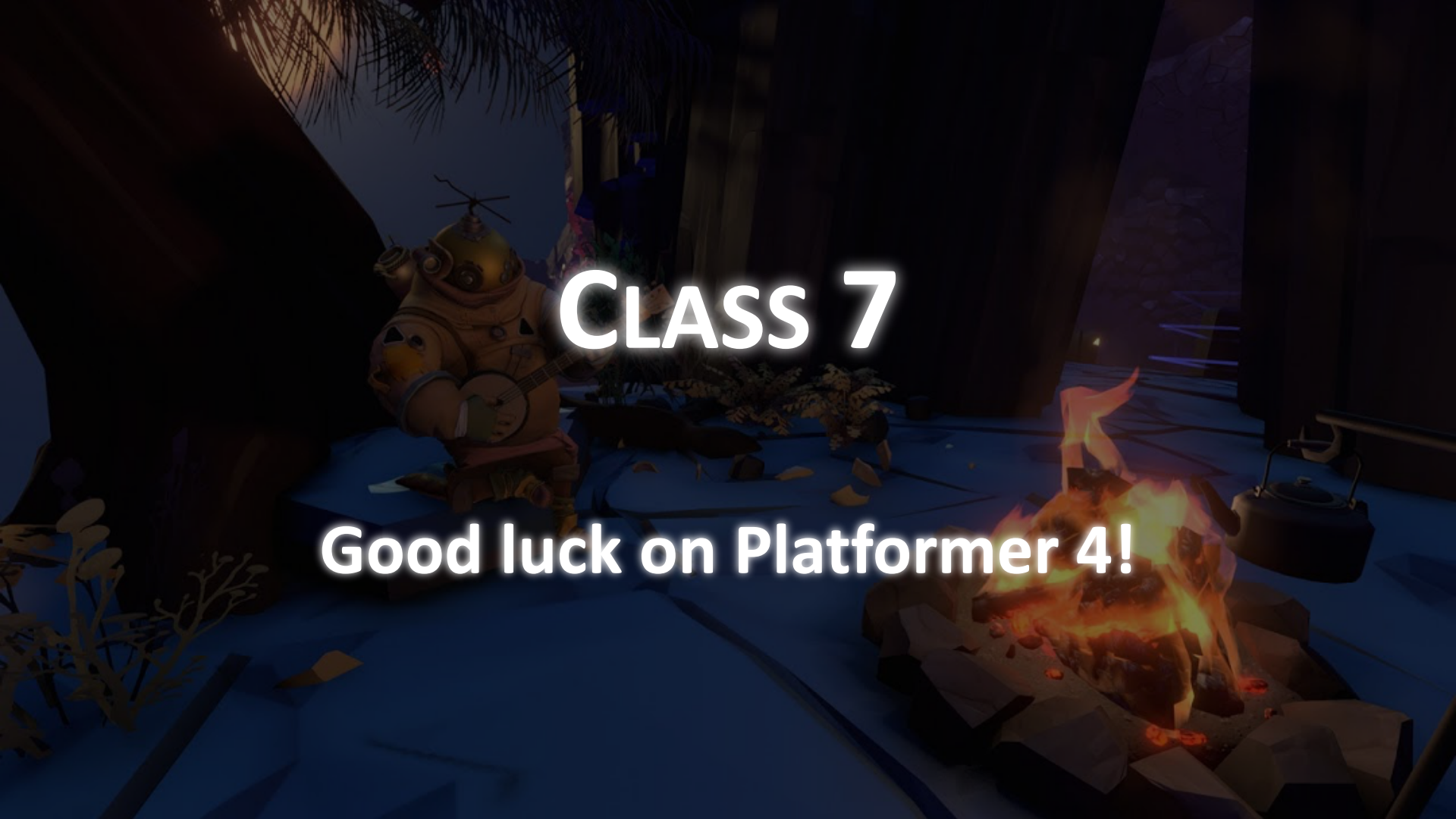
Linear Blend

Dual Quaternion

Ragdoll Physics

- Simulate parts of a character as constrained rigid bodies




Class 7
Good luck on Platformer 4!