https://gitlab.com/mhartfield/skinning-and-mesh-interactivity

Report for A4: Skinning and Mesh Interactivity

We, Raymond Wang and Max Hartfield, have fully completed what was required for the Skinning and Mesh Interactivity project. We have implemented Bone Picking, Updating the Skeleton, Linear Blend Skinning, and Texture Mapping.

Skeleton Hierarchy:

We implemented the Skeleton Hierarchy based on the pre-implemented classes for bone and mesh. The hierarchy is defined through the mesh's bone array as well as each bone's children array of bone indexes. This avoided having to make a separate / new bone class when the pre-provided one was sufficient enough for hierarchical traversal.

Bone Picking:

Mouse coordinates converted to world space: This was done in a screenToWorld method that generated a ray by unproject (x, y, 0) to get the world space coordinates and subtracting the eye position to get a world space ray.

```
public ScreenToWorld(x: number, y: number): Vec3 {
   // Unproject the screen coordinates (x, y, 0) to get the world space coordinates
   // https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/gluUnProject.xml
   // Assuming model matrix is identity?
   let objX: number =  2 * x / this.width - 1;
   let objY: number = 1 - 2 * y / this.viewPortHeight;
   let objZ: number = -1;

   let res: Vec4 = this.projMatrix().inverse(new Mat4()).multiplyVec4(new Vec4([objX,
objY, objZ, 1]));
   res = res.scale(1 / res.w);
   res = this.viewMatrix().inverse(new Mat4()).multiplyVec4(res);
   res = res.scale(1 / res.w)
   return new Vec3(res.xyz).subtract(this.camera.pos(), new Vec3()).normalize();
 }
```

Bone picking using cylinder intersect: This was done by aligning the bone along the y axis and doing the same transformation to the generated ray. We also shifted the ray to the bone's origin. This effectively turned the problem into a 2D problem so we just used the formula for a circle and quadratic formula to get the two t values. If the t values are greater than zero and the intersection point is within the height of the cylinder then we know we have intersected. We looped through all the bones and selected the bone with the minimum t value.

```
let world: Vec3 = this.ScreenToWorld(x, y);
     let minT : number = 10000;
     let minIndex : number = -1;
     //update scale translate and rotate
     for(var i: number = 0; i < this.animation.getScene().meshes[0].bones.length; i++)
{
       //translate ray to bones local coordinates which is aligned on y axis
       let bone : Bone = this.animation.getScene().meshes[0].bones[i];
       let height: number = bone.endpoint.subtract(bone.position, new
Vec3()).length();
       let boneDir : Vec3 = Vec3.direction(bone.endpoint, bone.position);
       let theta : number = Math.acos(Vec3.dot(new Vec3([0, 1, 0]), boneDir));
       let rot : Mat3 = Mat3.identity.rotate(theta, Vec3.cross(boneDir, new
Vec3([0,1,0])), new Mat3());
       let rayPos : Vec3 = this.camera.pos().subtract(bone.position, new Vec3());
       let rayDir : Vec3 = world.copy();
       if(!(boneDir.x == 0 && boneDir.y == 1 && boneDir.z == 0)) {
         rayDir = rot.multiplyVec3(world);
         rayPos = rot.multiplyVec3(rayPos);
       }

       let a : number = Math.pow(rayDir.x, 2) + Math.pow(rayDir.z, 2)
       let b : number = 2 * (rayDir.x * rayPos.x + rayDir.z * rayPos.z);
       let c : number = Math.pow(rayPos.x, 2) + Math.pow(rayPos.z, 2) - Math.pow(.07,
2);
       let discriminant: number = Math.pow(b, 2) - 4 * a * c;
       if(discriminant >= 0) {
         //quadratic formula
         let t1: number = (-1 * b + Math.pow(discriminant, .5)) / (2 * a);
         let t2: number = (-1 * b - Math.pow(discriminant, .5)) / (2 * a);
         if(t1 >= 0) {
           let intersection : Vec3 = rayPos.add(rayDir.scale(t1, new Vec3()), new
Vec3());
           if(intersection.y >= 0 && intersection.y <= height) {
             if(t1 < minT) {
               minT = t1;
               minIndex = i;
             }
           }
         }
         if(t2 >= 0) {
           let intersection : Vec3 = rayPos.add(rayDir.scale(t2, new Vec3()), new
Vec3());
           if(intersection.y >= 0 && intersection.y <= height) {
             if(t2 < minT) {
               minT = t2;
               minIndex = i;
             }
           }
         }
       }
     }
     this.selectedBone = minIndex;
```

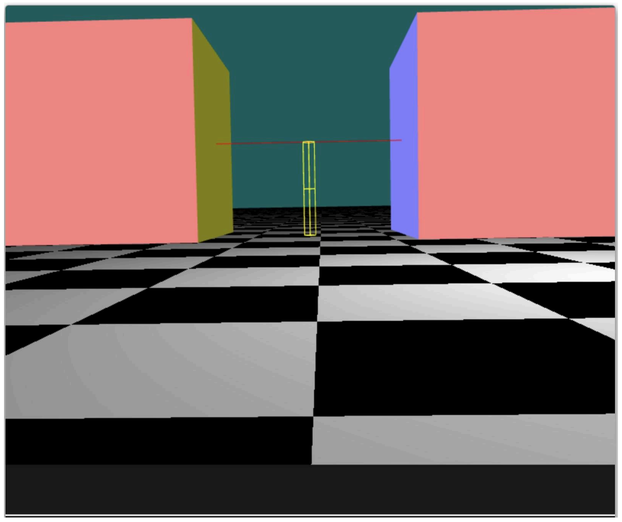
Bone visualization as cylinders: We created the highlight cylinder by first making a unit cylinder using the parametric equations for a circle, sin and cos, and a line render pass. Then, we applied a rotation and translation to align the cylinder with the selected bone in a render pass we made specifically for the cylinder.

```
//TODO: Create functionality for bone manipulation/key-framing
public getHighlightIndices(): Uint32Array {
  let arr : number[] = [];
  for(let i : number = 0; i < 48; i++){
    arr.push(i);
  }

  return new Uint32Array(arr);
}

public getHighlightPositions(): Float32Array {
  //unit cylinder, change height in future to match height of bone
  return new Float32Array(this.getCylinder(0.07, 6.0, 1.0))
}

public getCylinder(radius : number, n : number, height: number) {

  let arr : number[] = [];
  let theta : number = 2.0 * Math.PI / n;

  for (var i : number = 0; i < n; i++){
    let cosI : number = radius * Math.cos(theta * i);
    let cosI1 : number = radius * Math.cos(theta * (i + 1.0));
    let sinI : number = radius * Math.sin(theta * i);
    let sinI1 : number = radius * Math.sin(theta * (i + 1.0))

    arr.push(
      // bottom ring
      cosI, 0.0, sinI,
      cosI1, 0.0, sinI1,

      // middle ring
      cosI, (height / 2), sinI,
      cosI1, (height / 2), sinI1,

      // top ring
      cosI, height, sinI,
      cosI1, height, sinI1,

      // connecting line
      cosI, 0.0, sinI,
      cosI, height, sinI
    )
  }
  return arr
}
```

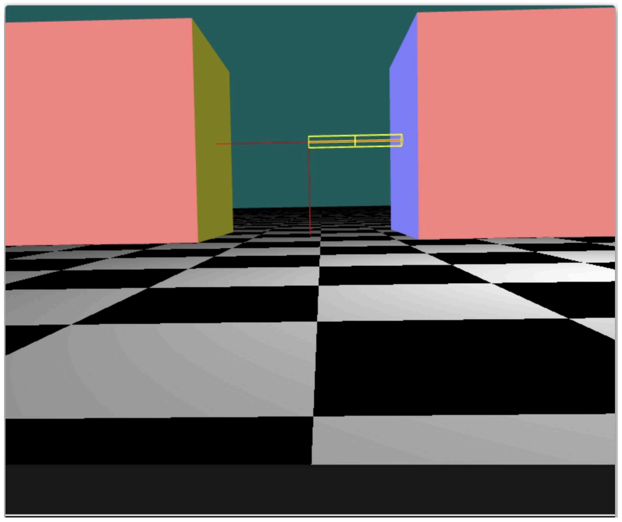
```
public initHighlight(): void {

this.highlightRenderPass.setIndexBufferData(this.scene.meshes[0].getHighlightIndices()
);
   this.highlightRenderPass.addAttribute("vertPosition", 3, this.ctx.FLOAT, false,
     3 * Float32Array.BYTES_PER_ELEMENT, 0, undefined,
this.scene.meshes[0].getHighlightPositions());
   //add uniform for scale, translate, and rotate
   this.highlightRenderPass.addUniform("uWorld",
     (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
       gl.uniformMatrix4fv(loc, false, new Float32Array(Mat4.identity.all()));
   });
   this.highlightRenderPass.addUniform("uProj",
     (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
       gl.uniformMatrix4fv(loc, false, new Float32Array(this.gui.projMatrix().all()));
   });
   this.highlightRenderPass.addUniform("uView",
     (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
       gl.uniformMatrix4fv(loc, false, new Float32Array(this.gui.viewMatrix().all()));
   });

   this.highlightRenderPass.addUniform("bRot",
     (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
       gl.uniformMatrix3fv(loc, false, this.gui.getBRot().all());
   });

   this.highlightRenderPass.addUniform("bScale",
   (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
     gl.uniformMatrix3fv(loc, false, this.gui.getBScale().all());
 });

   this.highlightRenderPass.addUniform("bTrans",
   (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
     gl.uniformMatrix4fv(loc, false, this.gui.getBTrans().all());
 });

   this.highlightRenderPass.addUniform("boneSelected",
     (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
         gl.uniform1fv(loc, [(this.gui.boneSelected() ? 1.0 : 0.0)]);
 });
   this.highlightRenderPass.setDrawData(this.ctx.LINES,
this.scene.meshes[0].getHighlightIndices().length, this.ctx.UNSIGNED_INT, 0);
   this.highlightRenderPass.setup();
 }
```

Pictures for Bone Picking:

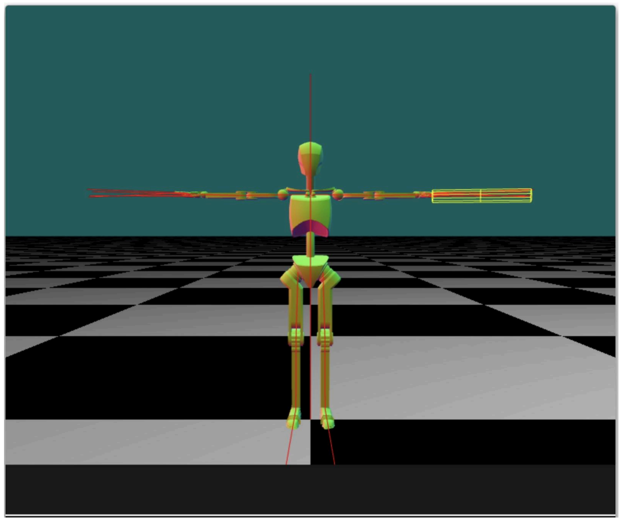
Mouse on bottom bone:



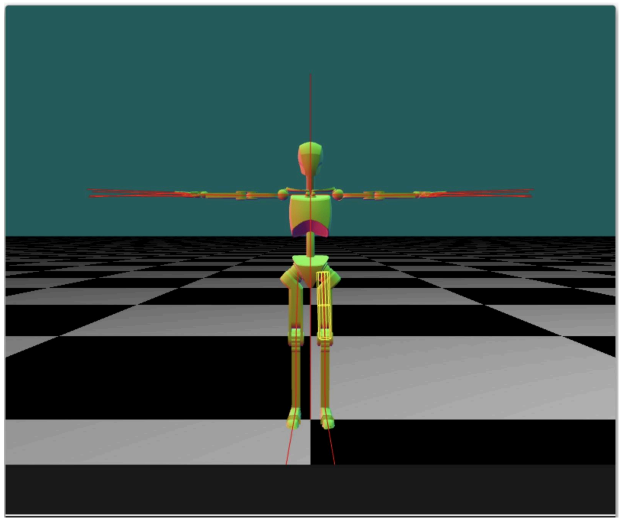
Mouse on right bone:



Mouse on far right hand bone:



Mouse on right thigh bone:



Updating the Skeleton:

Rotations apply to bone and child bones: Once we figured out the logic for rotating the parent bone, we propagated the quaternion and position to be rotated around through all the children bones recursively to update the rest of the skeleton.

```
public updateBones(bone : Bone, rot : Quat, pos : Vec3) {
   if(bone.children.length == 0){
     return;
   }
   for(let i : number = 0; i < bone.children.length; i++){
     let tempBone : Bone =
this.animation.getScene().meshes[0].bones[bone.children[i]];
     tempBone.rotation = Quat.product(rot, tempBone.rotation, new Quat());
     tempBone.position = rot.multiplyVec3(tempBone.position.subtract(pos, new
Vec3())).add(pos, new Vec3());
     tempBone.endpoint = rot.multiplyVec3(tempBone.endpoint.subtract(pos, new
Vec3())).add(pos, new Vec3());
     this.updateBones(tempBone, rot, pos);
   }
 }
```

Bone rotates relative to camera coordinate system: Implementing the parts for updating the skeleton was pretty challenging. We ended up implementing the rotation of bones as follows: First, we projected the selected bone’s position and endpoint into 2d NDC coords by applying the view and projection matrix to each vector and then only taking the x and y components of the resulting vector. Then, we also calculated the mouse’s 2d NDC coordinates. We can then construct the two vectors from the position to endpoint as well as from position to mouse point in NDC. By using the dot product and arccos, we are able to get the angle, theta, between the two vectors. Because arccos is bounded, we then constructed a 2x2 rotation matrix to rotate the first vector by. If the rotation applied to the first vector accurately rotated it to the second, then we kept the found theta value. Otherwise, we negated the theta value. Finally, we applied the rotation to the skeleton by constructing a quaternion from the look axis and theta angle, rotating the selected bone’s endpoint and updating its quaternion.

```
// rotate the bone
           let bone : Bone =
this.animation.getScene().meshes[0].bones[this.selectedBone];
           let rotAxis: Vec3 = this.camera.forward().normalize(new Vec3());

           let pos : Vec4 = new Vec4([bone.position.x, bone.position.y,
bone.position.z, 1.0]);
           let endpoint : Vec4 = new Vec4([bone.endpoint.x, bone.endpoint.y,
bone.endpoint.z, 1.0])
           let screen : Vec4 = this.viewMatrix().multiplyVec4(pos, new Vec4());
           let screenend : Vec4 = this.viewMatrix().multiplyVec4(endpoint, new
Vec4());
           screen.scale(1 / screen.w);
           screenend.scale(1 / screenend.w);

           let ndcPos : Vec4 = this.projMatrix().multiplyVec4(screen, new Vec4());
           let ndcEnd : Vec4 = this.projMatrix().multiplyVec4(screenend, new Vec4());
           ndcPos.scale(1 / ndcPos.w);
           ndcEnd.scale(1 / ndcEnd.w);

           let pos2d : Vec2 = new Vec2(ndcPos.xy);
           let end2d : Vec2 = new Vec2(ndcEnd.xy);
           let mouse2d : Vec2 = new Vec2([((x / this.width) * 2) - 1, 1 - ((y / this.viewPortHeight) * 2)]);
           // console.log("POS: " + pos2d.xy + " END: " + end2d.xy + " Mouse: " + mouse2d.xy);

           let endVec : Vec2 = end2d.subtract(pos2d, new Vec2()).normalize();
           let mouseVec : Vec2 = mouse2d.subtract(pos2d, new Vec2()).normalize();
           // console.log("endVec: " + endVec.xy + " mouseVec: " + mouseVec.xy);
           let dot: number = Vec2.dot(endVec, mouseVec);
           if(dot > 1) {
             dot = 1;
           } else if(dot < -1) {
             dot = -1;
           }
           let angle : number = Math.acos(dot);
           // let angle : number = Math.acos(Vec2.dot(pos2d.normalize(), mouse2d.normalize()));
           let rot2 : Mat2 = new Mat2([Math.cos(angle), Math.sin(angle), -1 * Math.sin(angle), Math.cos(angle)]);

           let theta : number;
           // console.log("ROT END: " + Math.round(rot2.multiplyVec2(end2d, new Vec2()).normalize().x * 100) / 100 + " " + Math.round(rot2.multiplyVec2(end2d, new Vec2()).normalize().y * 100) / 100 + " NORMMOUSE: " + Math.round(mouse2d.normalize(new Vec2()).x * 100) / 100 + " " + Math.round(mouse2d.normalize(new Vec2()).y * 100) / 100);

           if(rot2.multiplyVec2(endVec, new Vec2()).subtract(mouseVec).length() < .001) {
             theta = angle;
           } else {
             theta = -1 * angle;
           }

           let rot : Quat = Quat.fromAxisAngle(rotAxis, theta);
           bone.rotation = Quat.product(rot, bone.rotation, new Quat());
           bone.endpoint = rot.multiplyVec3(bone.endpoint.subtract(bone.position, new Vec3())).add(bone.position, new Vec3());
```

```
          // console.log("ROT: " + bone.rotation.xyzw + " END: " + bone.endpoint.xyz
);

          this.updateBones(bone, rot, bone.position.copy());
```

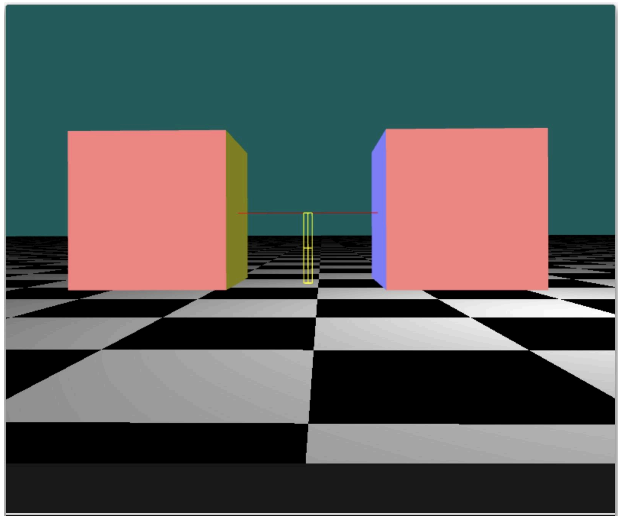
Left and right arrow keys roll bone when selected: This was pretty easy to implement since we already understood the logic for rotating the bone. It was similar except the axis is now the bone’s tangent. We found the correct quaternion for the rotation with the axis as the endpoint - position vector and the angle as the roll speed. Then, we applied the rotation and propagated it to the child bones.

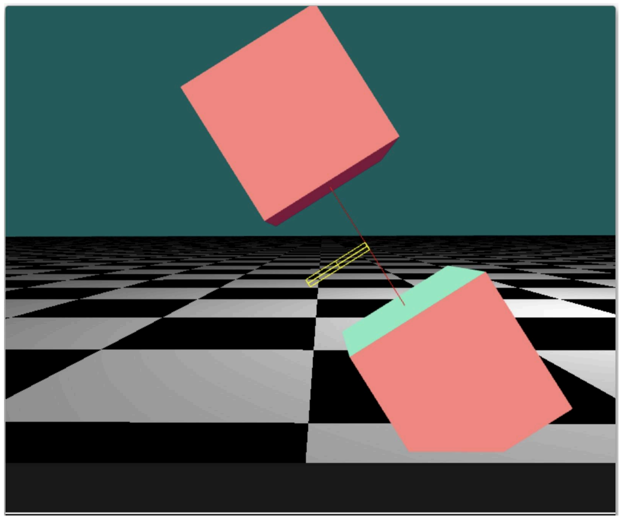
```
case "ArrowLeft": {
       //TODO: Handle bone rolls when a bone is selected
      if(this.boneSelected()) {
          let bone: Bone =
this.animation.getScene().meshes[0].bones[this.selectedBone]
          let rotAxis: Vec3 = bone.endpoint.subtract(bone.position, new
Vec3()).normalize();
          let rot : Quat = Quat.fromAxisAngle(rotAxis, -1.0 * GUI.rollSpeed);
          bone.rotation = Quat.product(rot, bone.rotation, new Quat());
          this.updateBones(bone, rot, bone.position.copy())
        } else {
          this.camera.roll(GUI.rollSpeed, false);
        }
        break;
     }
     case "ArrowRight": {
        //TODO: Handle bone rolls when a bone is selected
        if(this.boneSelected()) {
          let bone: Bone =
this.animation.getScene().meshes[0].bones[this.selectedBone]
          let rotAxis: Vec3 = bone.endpoint.subtract(bone.position, new
Vec3()).normalize();
          let rot : Quat = Quat.fromAxisAngle(rotAxis, GUI.rollSpeed);
          bone.rotation = Quat.product(rot, bone.rotation, new Quat());
          this.updateBones(bone, rot, bone.position.copy())
        } else {
          this.camera.roll(GUI.rollSpeed, true);
        }
        break;
     }
```

Pictures for Updating Bones:

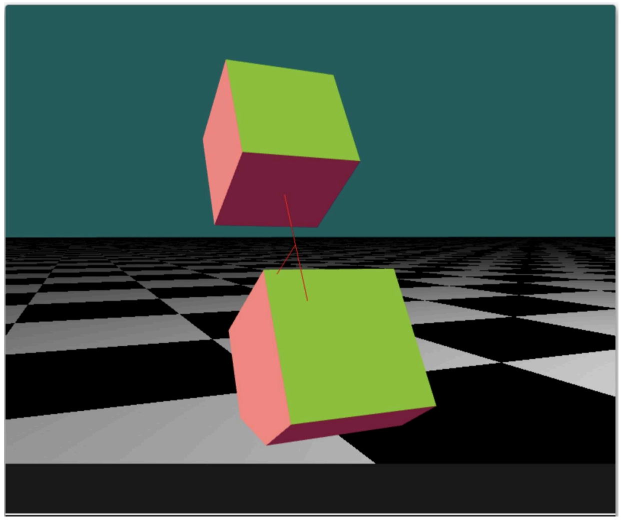
Bone Rotation:

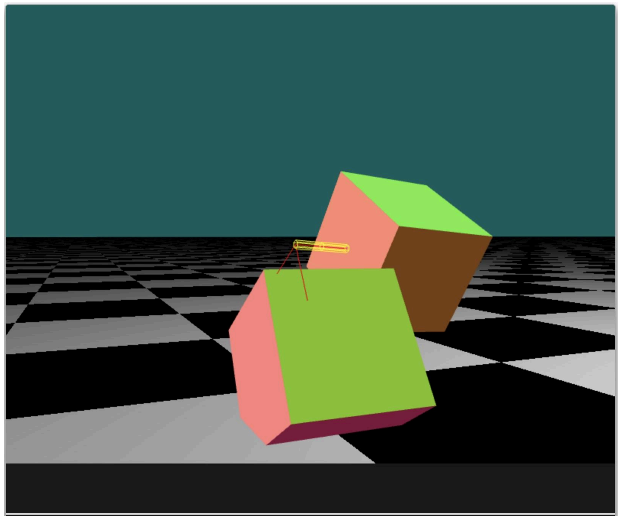
Rotating Parent Bone: Before After



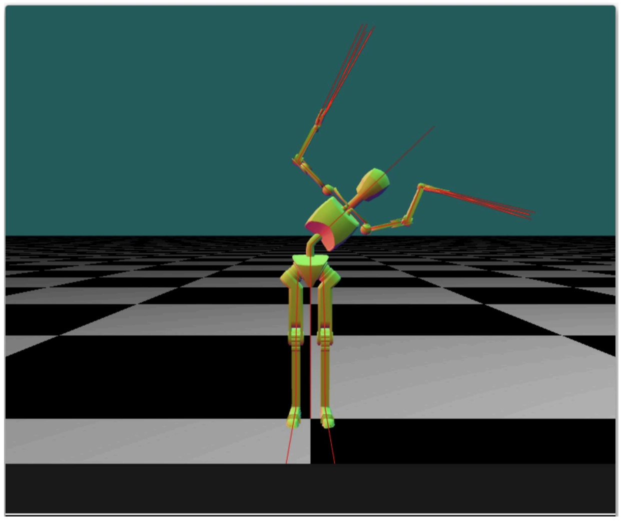
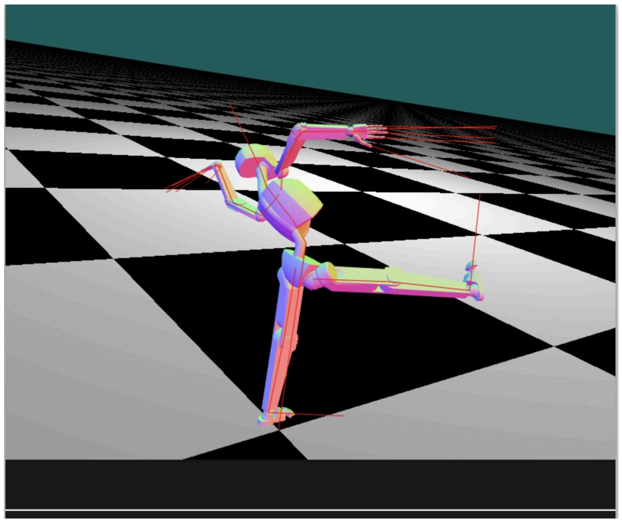


Rotating Child Bone: Before After



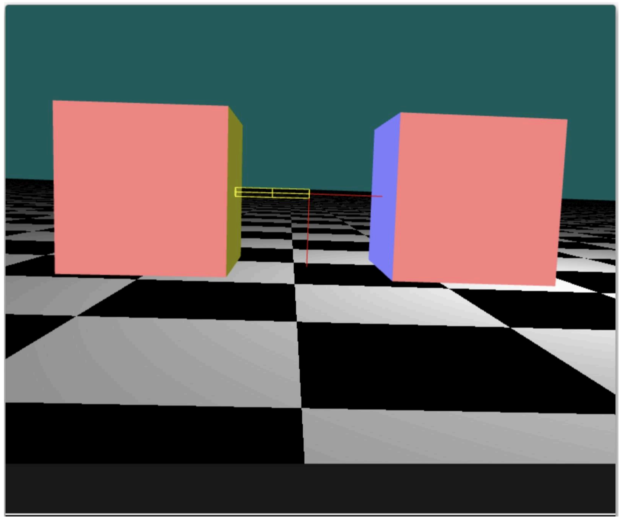
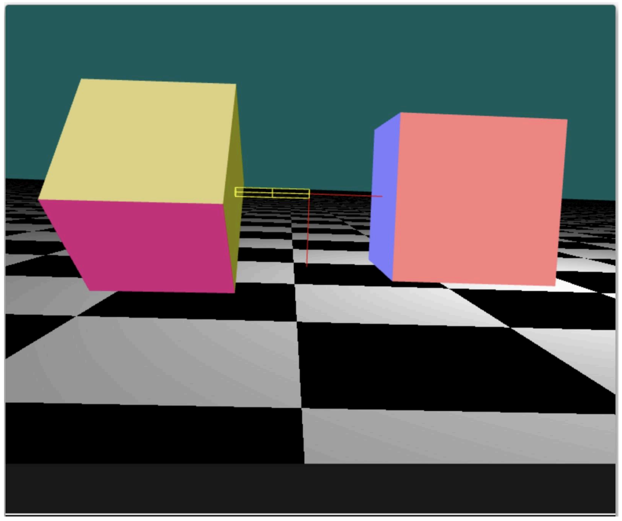


Rotating Skeleton: At First Cam Pos At Second Cam Pos

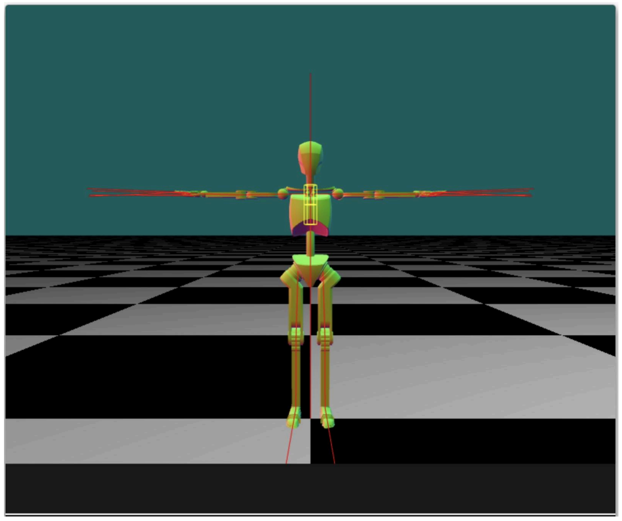
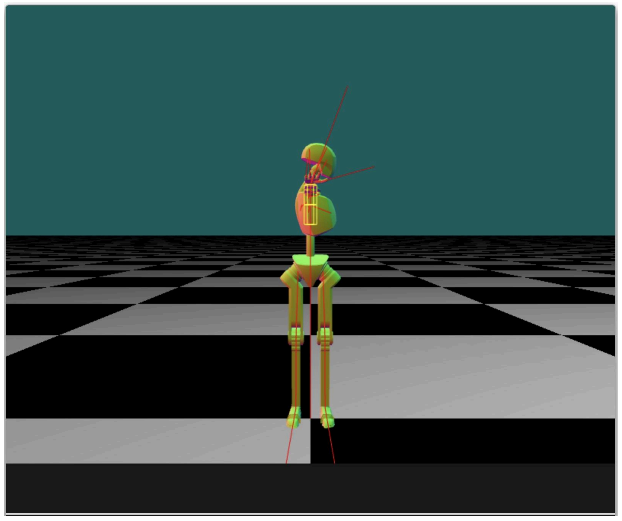



Bone Rolling:

Rolling Child Bone: Before After

Rolling Skeleton: Before After

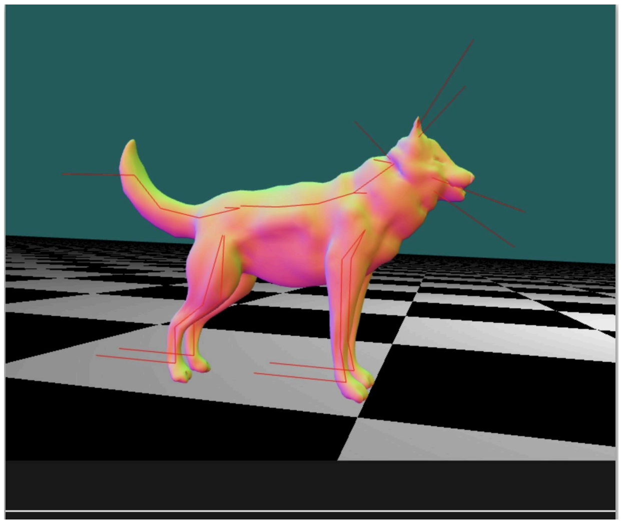
Linear Blend Skinning:
Linear blend skinning deforms mesh based on skeleton orientation: This was done in the scene vertex shader. I used v0, v1, v2, v3 as the vertex positions and rotated and translated it based on the respective jTrans and jRots. Then we summed the weight of each bone times the vertex position within that bone's coordinate system. For the normals, we did the same thing except we didn't use jTrans.

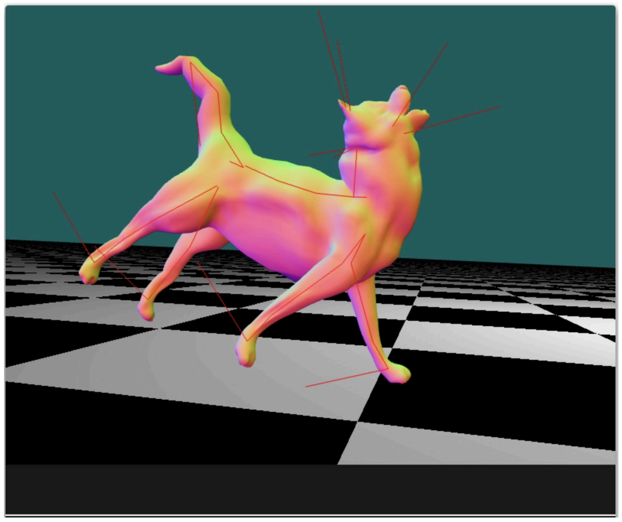
```
void main () {
       vec3 trans = vertPosition;
       vec3 bonePosition = (jTrans[int(skinIndices[0])] +
qtrans(jRots[int(skinIndices[0])], v0.xyz).xyz) * skinWeights[0];
       bonePosition += (jTrans[int(skinIndices[1])] +
qtrans(jRots[int(skinIndices[1])], v1.xyz).xyz) * skinWeights[1];
       bonePosition += (jTrans[int(skinIndices[2])] +
qtrans(jRots[int(skinIndices[2])], v2.xyz).xyz) * skinWeights[2];
       bonePosition += (jTrans[int(skinIndices[3])] +
qtrans(jRots[int(skinIndices[3])], v3.xyz).xyz) * skinWeights[3];

       vec4 worldPosition = mWorld * vec4(bonePosition, 1.0);
       gl_Position = mProj * mView * worldPosition;

       //  Compute light direction and transform to camera coordinates
       lightDir = lightPosition - worldPosition;

       vec4 aNorm4 = vec4(aNorm, 0.0);
       vec3 skinnedNormal = (qtrans(jRots[int(skinIndices[0])], aNorm).xyz) *
skinWeights[0];
```

```
       skinnedNormal += (qtrans(jRots[int(skinIndices[1])], aNorm).xyz) *
skinWeights[1];
       skinnedNormal += (qtrans(jRots[int(skinIndices[2])], aNorm).xyz) *
skinWeights[2];
       skinnedNormal += (qtrans(jRots[int(skinIndices[3])], aNorm).xyz) *
skinWeights[3];
       normal = normalize(mWorld * vec4(skinnedNormal, 0.0));
       uv = aUV;
   }
```

Pictures of Linear Blend Skinning:

Wolf Rotations: Before After





Texture Mapping:

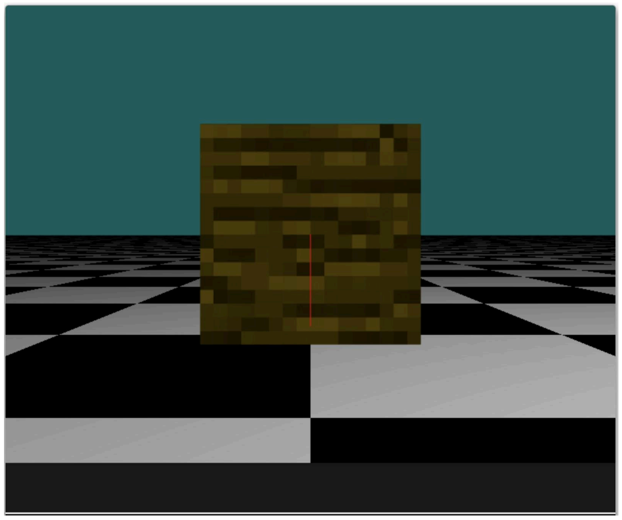
Added shader for texture mapping: This was done in the scene fragment shader. If there was a texture map I calculated the respective texColor and applied it to the original color.

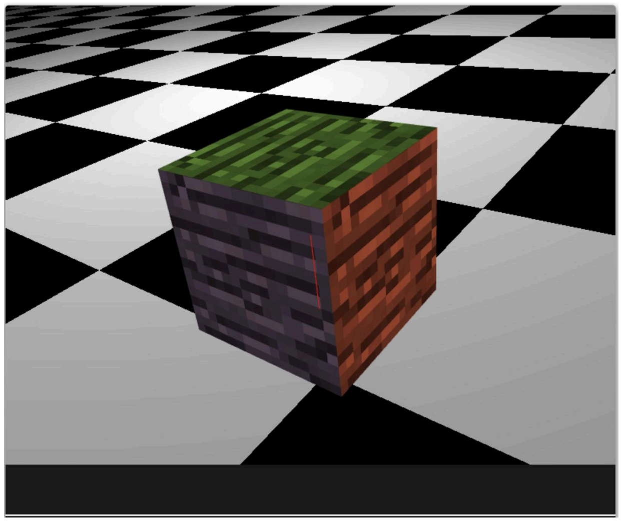
```
export const sceneFSText = `
   precision mediump float;

   varying vec4 lightDir;
   varying vec2 uv;
   varying vec4 normal;

   uniform sampler2D sampler2d;
   uniform float textureMap;

   void main () {
       vec4 texColor = vec4(1.0, 1.0, 1.0, 1.0);
       if(textureMap == 1.0) {
           texColor = texture2D(sampler2d, uv);
       }
       gl_FragColor = vec4((normal.x + 1.0)/2.0, (normal.y + 1.0)/2.0, (normal.z +
1.0)/2.0,1.0) * texColor;
   }
`;
```

I also added the textureMap to the render pass.

```
if(this.scene.meshes[0].imgSrc) {
     this.sceneRenderPass.addTextureMap(this.scene.meshes[0].imgSrc)
     this.sceneRenderPass.addUniform("sampler2D",
     (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
         gl.uniform1i(loc, 0);
     });
   }
   this.sceneRenderPass.addUniform("textureMap",
   (gl: WebGLRenderingContext, loc: WebGLUniformLocation) => {
       gl.uniform1fv(loc, [(this.scene.meshes[0].imgSrc ? 1.0 : 0.0)]);
   });
   this.sceneRenderPass.setup();
```

Pictures of Texture Mapping:





Issues we encountered and fixed:

Bone Picking: One issue that we faced and fixed in bone picking was just getting the unit cylinder to render properly. This was caused by us using triangles instead of lines and thus trying to render a cylinder and then using a shader to select the endpoints, instead of just using lines, which fixed the issue.

Updating the Skeleton: Earlier on in the development, we were encountering an issue where the rotating bone would rotate erratically and jump around. We fixed this by realizing that we should be constructing the two vectors to calculate the rotation instead of just using the endpoint and the mouse position vectors. Additionally, we encountered a problem where the 2x2 rotation matrix when applied was never getting close to the endpoint vector and was always negating the angle, resulting in a bone that could only rotate along half of 360 degrees. We fixed this by figuring out we needed to normalize the vectors that were to be rotated. Finally, another issue that we faced was that after rotating a certain time, the rotation would become a NaN and then the bone would disappear. We found this was caused by arcos getting passed a value of 1.000001 because of floats and fixed this by capping it at 1.

Linear Blend Skinning: We had an issue where the colors were not changing properly. Through asking on discord we found out that we are not supposed to apply the jTrans to the aNorm, just the jRots.

Texture Mapping: When switching from something that had the texture map to something that didn't, the texture map would stay. To fix this, we created a uniform called textureMap that represented whether a textureMap was in use or not.

Known Bugs:
We have no known bugs.

Future Work:
We have completed all required aspects of this project and have no future work to do.