Skinning

Rohil Verma: rbv299
Cliff Xu: cx2356
Github Link:

Bone Picking:

To start bone picking, we created a screenToWorld function that “unprojects” a ray from eye to mouse to create a ray.

```
public screenToWorld(x: number, y: number): Vec3 {
  let ndcX = 2 * x / this.width - 1
  let ndcY = 1 - (2 *y / this.viewPortHeight)
  let ndc = new Vec4([ndcX, ndcY, -1, 1])
  let projInverse = this.projMatrix().copy().inverse()
  let unproject = projInverse.multiplyVec4(ndc)
  unproject.w = 0
  let viewInv = this.viewMatrix().copy().inverse()
  let dir = viewInv.multiplyVec4(unproject)
  let rayDirection = new Vec3([dir.x, dir.y, dir.z])
  return rayDirection
}
```

Then for cylinder intersection, we have a function that takes the world ray, projects it into the local space of bones/cylinders and checks if it intersects with the circle using a quadratic equation. If the ray intersects, check if its within the bounds of the height of the cylinder.

```
rayCylinderIntersection(rayOrigin: Vec3, rayDirection: Vec3, radius: number, height: number): number {
  // Normalize the ray direction
  const normalizedRayDirection = rayDirection.copy().normalize()

  const a = normalizedRayDirection.x * normalizedRayDirection.x + normalizedRayDirection.z * normalizedRayDirection.z;
  const b = 2 * (rayOrigin.x * normalizedRayDirection.x + rayOrigin.z * normalizedRayDirection.z);
  const c = rayOrigin.x * rayOrigin.x + rayOrigin.z * rayOrigin.z - radius * radius;

  const discriminant = b * b - 4 * a * c;

  if (discriminant < 0) {
    return NaN;
  }

  const t1 = (-b - Math.sqrt(discriminant)) / (2 * a);
  const t2 = (-b + Math.sqrt(discriminant)) / (2 * a);

  const y1 = rayOrigin.y + t1 * normalizedRayDirection.y;
  const y2 = rayOrigin.y + t2 * normalizedRayDirection.y;

  let tMin = Number.MAX_VALUE;

  if (y1 >= 0 && y1 <= height) {
    tMin = Math.min(tMin, t1);
  }

  if (y2 >= 0 && y2 <= height) {
    tMin = Math.min(tMin, t2);
  }

  if (tMin === Number.MAX_VALUE) {
    return NaN;
  }

  return tMin;
}
```

For cylinder generation, we create vertices for the respective cylinder and transform them into the correct position using a TNB matrix.

```
// Create the top hexagon
for (let j = 0; j < 6; j++) {
  const angle = j * angleIncrement;
  const x = this.kCylinderRadius * Math.cos(angle);
  const z = this.kCylinderRadius * Math.sin(angle);
  let verts = new Vec3([x, length, z])

  verts = verts.multiplyMat3(tnb)
  vertices.push(verts.x + startPos.x, verts.y + startPos.y, verts.z + startPos.z);
}

// Create the bottom hexagon
for (let j = 0; j < 6; j++) {
  const angle = j * angleIncrement;
  const x = this.kCylinderRadius * Math.cos(angle);
  const z = this.kCylinderRadius * Math.sin(angle);
  let verts = new Vec3([x, 0, z])

  verts = verts.multiplyMat3(tnb)
  vertices.push(verts.x + startPos.x, verts.y + startPos.y, verts.z + startPos.z);
}
```

Updating the Skeleton:

To rotate the bone, we first get the axis of rotation and create a quaternion using that and the rotation speed. The quaternion is then applied to the bone's rotation quaternion and its endpoints and is propagated down all the subsequent children.

```
public rotateBone(bone: Bone, rotQuat: Quat): void {
  bone.rotation = Quat.product(rotQuat, bone.rotation);
  let originalEnd = bone.endpoint.copy()
  let end = bone.endpoint.copy();
  let relativeEnd = Vec3.difference(end, bone.position);
  let endQuat = new Quat([relativeEnd.x, relativeEnd.y, relativeEnd.z, 0.0]);
  let endCalc = Quat.product(Quat.product(rotQuat, endQuat), rotQuat.copy().conjugate());
  bone.endpoint.x = endCalc.x + bone.position.x;
  bone.endpoint.y = endCalc.y + bone.position.y;
  bone.endpoint.z = endCalc.z + bone.position.z;
  let difference = Vec3.difference(bone.endpoint, end);
  for (let child = 0; child < bone.children.length; child++) {
    let childIndex = bone.children[child];
    let childBone = this.animation.getScene().meshes[0].bones[childIndex];
    this.moveChild(childBone, difference, rotQuat, originalEnd.copy());
  }
}

public rotateWithQuat(quat: Quat, vec: Vec3): Vec3 {
  const vecQuat = new Quat([vec.x, vec.y, vec.z, 0.0]);
  const resultQuat = Quat.product(Quat.product(quat, vecQuat), quat.copy().conjugate());
  return new Vec3([resultQuat.x, resultQuat.y, resultQuat.z]);
}
```

```
public moveChild(bone: Bone, translation: Vec3, rotQuat: Quat, parentEnd: Vec3): void {

  let rotatedPosition = this.rotateWithQuat(rotQuat, Vec3.difference(bone.position, parentEnd)).add(parentEnd.copy());
  let rotatedEndpoint = this.rotateWithQuat(rotQuat, Vec3.difference(bone.endpoint, parentEnd)).add(parentEnd.copy());

  let originalEnd = bone.endpoint.copy()

  bone.position = rotatedPosition.add(translation);
  bone.endpoint = rotatedEndpoint.add(translation);

  bone.rotation = Quat.product(rotQuat, bone.rotation);

  for (let child = 0; child < bone.children.length; child++) {
    let childIndex = bone.children[child];
    let childBone = this.animation.getScene().meshes[0].bones[childIndex];
    this.moveChild(childBone, translation, rotQuat, parentEnd.copy());
  }
}
```

The same can be applied to rolling except the axis of rotation is the bone's axis instead of the look direction.

Linear Blend Skinning:
This is done in the sceneVSText shader by applying the qtrans method from the skeletonVS shader jTrans and jRots for each of the skin indices and their respective weights.

```
vec3 qtrans(vec4 q, vec3 v) {
    return v + 2.0 * cross(cross(v, q.xyz) - q.w*v, q.xyz);
}

void main () {

    vec3 trans = vec3(0, 0, 0);
    for(int i = 0; i < 4; i++){

        if (i == 0){
            v = v0;
        } else if (i == 1){
            v = v1;
        } else if (i == 2){
            v = v2;
        } else {
            v = v3;
        }

        index = int(skinIndices[i]);
        weight = float(skinWeights[i]);
        vec3 local = jTrans[index] + qtrans(jRots[index], v.xyz);
        trans.x = trans.x + (weight * local.x);
        trans.y = trans.y + (weight * local.y);
        trans.z = trans.z + (weight * local.z);

    }
```

Texture Mapping:
In texture mapping, we checked if there was a texture map loaded in. If there was, we added that map to the render pass. In the fragment shader, we used the uniform sampler and the texture2D method to get the color we want based on the texture map and aUV coordinates.

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

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

    uniform sampler2D ourTexture;

    void main () {
        gl_FragColor = texture2D(ourTexture, uv);
    }
`;
```

```
if(this.scene.meshes[0].imgSrc){
  this.sceneRenderPass.addTextureMap(this.scene.meshes[0].imgSrc, textureVSText, textureFSText);
}
```

Some Screenshots: