20170058 **Keonwoo Kim** · June 12, 2021
CS580 Computer Graphics

Term Project Implementation Report

Contents

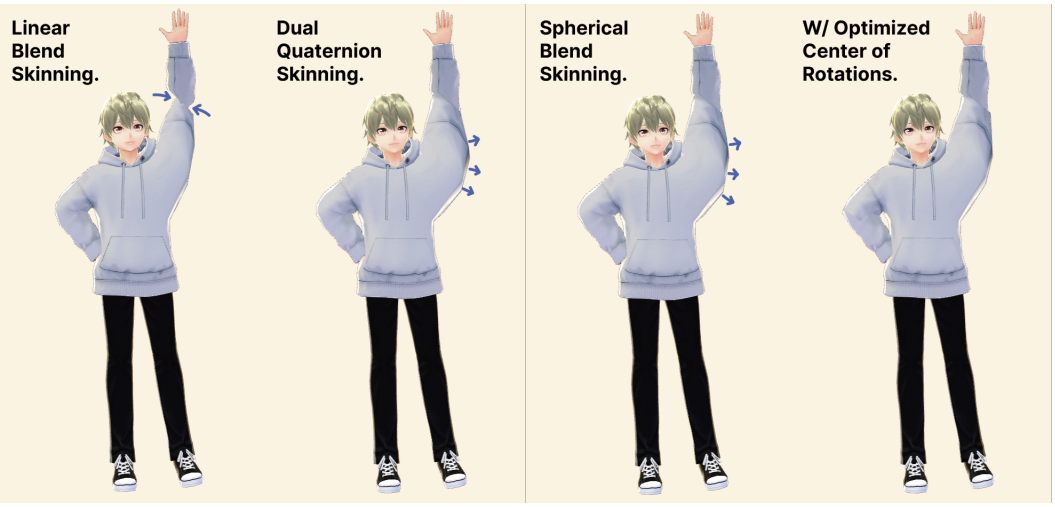

Figure 1: The comparison of skinning methods.

0 Introduction

The project demonstrates various real-time skinning algorithms: linear blend skinning (LBS), dual quaternion skinning (DQS, or DLB) [KCŽO08], spherical blend skinning (SBS) [KŽ05], and a variant of spherical blend skinning with optimized center of rotations (CoR) [LH16].

The frontend web app is written mainly in vanilla TypeScript and bundled by webpack. The backend server app is written in TypeScript using *fastify*.

1 Project Structure

1.1 Frontend

The abstract class for an app is in `src/App/Types`, and it is almost the same with one of `cs580` module. This app is created by the Redux reducer (`src/Store/Reducer/App`) when the app state is changed from `NONE` to `MAIN` (`src/index:6`.) Backend server status is managed by the reducer (`src/Store/Reducer/Server`.)

The shaders used to skin vertices are defined in `src/Shader/*`. `substi tuteShader` replaces `#include<...>` directives by corresponding shader chunks until there are no replacements anymore. Common shader chunks are located in `src/Shader/Chunk`, and other shader-specific chunks are in `src/Shader/*/Chunk`.

`wasm-blend` contains WebAssembly files providing SBS centers of rotation data, while it is not used in the project because it seems there are no advantage in

performance over copying data from JS side to WASM side.

Static files including the model files are in `dist/static` directory.

1.2 Backend

The backend project is in `cor` directory. The default port number for the backend is `9001`. The routes defined on the server (`cor/src/server.ts`) are as follows:

`GET /`: To check server availability.

`static /cache/* -> /*`: To serve cached optimized center of rotations binary data used by CoR.

`POST /`: To receive requests to generate CoR binary data.

`GET /:modelName`: To check if CoR data are generated completely for a model `modelName`.

Workers that actually generate CoR data are in `cor/src/worker.ts`.

2 Load Models

The format of the default model I used in this project is VRM. VRM is based on glTF 2.0 and is specialized to represent humanoid models. To load VRM models into Three.js project using three-vrm, we need to load them as glTFs using GLTFLoader `three/examples/jsm/loaders/GLTFLoader` and parse as VRMs using `await VRM.from()`. To reduce time to traverse `SkinnedMesh`es and `ShaderMaterial`s in the model, they are stored in `MainApp.vrms`. Finally, it tries to fetch optimized CoR data by long polling (`MainApp.checkOptimizedCoRLongPolling()`.) When the data fetches are completed, CoR button on the side panel gets enabled.

3 Linear Blend Skinning

The default vertex shader for a VRM model is defined in three-vrm, and it uses linear blend skinning (`#include <skinning_vertex>`.) I mention here clearly that I did nothing to implement linear blend skinning, and the default shader implements the linear blend skinning.

```
// skinning_vertex.glsl
  vec4 skinVertex = bindMatrix * vec4( transformed, 1.0 );
  vec4 skinned = vec4( 0.0 );
  skinned += boneMatX * skinVertex * skinWeight.x;
  skinned += boneMatY * skinVertex * skinWeight.y;
  skinned += boneMatZ * skinVertex * skinWeight.z;
  skinned += boneMatW * skinVertex * skinWeight.w;
  transformed = ( bindMatrixInverse * skinned ).xyz;
```

4 Helper Shader Chunks

4.1 Quaternions

A quaternion is represented by a `vec4`. `quatToMat3/4` converts a quaternion into a rotational matrix.

```
// src/Shader/Chunk/quaternion.chunk.vert
mat3 quatToMat3( vec4 q ) {
  return mat3(
    // column 0
    1.0 - 2.0 * (q.y * q.y + q.z * q.z), 2.0 * (q.x * q.y + q.w * q.z),
    2.0 * (q.x * q.z - q.w * q.y),
    // column 1
    2.0 * (q.x * q.y - q.w * q.z),       1.0 - 2.0 * (q.x * q.x + q.z * q.z),
    2.0 * (q.y * q.z + q.w * q.x),
    // column 2
    2.0 * (q.x * q.z + q.w * q.y),       2.0 * (q.y * q.z - q.w * q.x),
    1.0 - 2.0 * (q.x * q.x + q.y * q.y)
  );
}
```

`getQLERP` interpolates 4 quaternions w.r.t. the given weights. Here `quatAntipodality` adjusts antipodality of quaternions, where $\mathbf{q}$ and $-\mathbf{q}$ represent the same 3D rotation while the interpolation results differ.

```
// src/Shader/Chunk/quaternion.chunk.vert
float quatAntipodality( vec4 q1, vec4 q2 ) {
  return dot( q1, q2 ) < 0.0 ? -1.0 : 1.0;
```

```
}

vec4 getQLERP(
  vec4 weights, vec4 quatX, vec4 quatY, vec4 quatZ, vec4 quatW
) {
  vec4 linearBlending = weights.x * quatX
      + weights.y * quatY * quatAntipodality( quatX, quatY )
      + weights.z * quatZ * quatAntipodality( quatX, quatZ )
      + weights.w * quatW * quatAntipodality( quatX, quatW );
  return linearBlending / length( linearBlending );
}
```

4.2 Dual Quaternions

A dual quaternion is represented by `struct DualQuat` containing two quaternions `rot` and `transl`. `getDLBMat4` blends 4 dual quaternions into one and converts it into a `mat4`. The algorithm is described in [KCŽO08, Algorithm 1].

```
// src/Shader/Chunk/dual_quaternion.chunk.vert
mat4 getDLBMat4(vec4 weights, DualQuat dqX, DualQuat dqY, DualQuat dqZ, DualQuat dqW
    ) {
  // b̂ in the paper
  DualQuat linearBlending;
  linearBlending.rot = /* omitted */;
  linearBlending.transl = /* omitted */;

  float linearBlendingNorm = length( linearBlending.rot );
  vec4 r = linearBlending.rot / linearBlendingNorm; // c0 in the paper
  vec4 e = linearBlending.transl / linearBlendingNorm; // cε in the paper
  return mat4( /* omitted */ );
}
```

5 Dual Quaternion Skinning

With `getDLBMat4` above, dual quaternion skinning is done very easily. First, bone matrices in dual quaternion form is given as uniform variables, packed in a `THREE.DataTexture`. The algorithm for `DualQuaternion.fromMat4` is described in [KCŽO08, Equation (25)].

```
// src/App/Main
switch (this.skinningType) {
case SkinningType.DUAL_QUATERNION: {
  const dualQuatBoneArray = _.range(mesh.skeleton.bones.length)
    .map((i) => DualQuaternion.fromMat4(
                  new THREE.Matrix4()
                    .fromArray(mesh.skeleton.boneMatrices, 16 * i)
                    .premultiply(mesh.bindMatrix)
                    .multiply(mesh.bindMatrixInverse)))
    .map((dq) => dq.toArray()).flat();

  const dataTexture = material.uniforms.dualQuatBoneTexture?.value
    as THREE.DataTexture;
  dataTexture.image.data.set(dualQuatBoneArray);
  dataTexture.needsUpdate = true;
  break;
}
```

```
// src/Shader/DualQuaternion/Chunk/dual_quaternion_skinnormal.chunk.vert
mat4 skinMatrix = getDLBMat4( skinWeight, boneDqX, boneDqY, boneDqZ, boneDqW );
objectNormal = vec4( skinMatrix * vec4( objectNormal, 0.0 ) ).xyz;
#ifdef USE_TANGENT
  objectTangent = vec4( skinMatrix * vec4( objectTangent, 0.0 ) ).xyz;
#endif
// src/Shader/DualQuaternion/Chunk/dual_quaternion_skinning.chunk.vert
transformed = ( skinMatrix * vec4( transformed, 1.0 ) ).xyz;
```

6 Spherical Blend Skinning

To implement spherical blend skinning, one needs to solve the following least squares problem for each vertex $\mathbf{v}$:

$$\text{minimize} \sum_{\substack{i<j \\ w_{\mathbf{v}}^{(i)} \neq 0 \neq w_{\mathbf{v}}^{(j)}}} \left\| (M^{(i)} - M^{(j)}) \begin{pmatrix} \mathbf{r}_{\mathbf{v}} \\ 1 \end{pmatrix} \Big|_{xyz} \right\|^2 = \sum_{\substack{i<j \\ w_{\mathbf{v}}^{(i)} \neq 0 \neq w_{\mathbf{v}}^{(j)}}} \| (R^{(i)} - R^{(j)}) \mathbf{r}_{\mathbf{v}} - (\mathbf{t}^{(j)} - \mathbf{t}^{(i)}) \|^2$$

$$= \left\| \underbrace{\begin{bmatrix} \vdots \\ R^{(i)} - R^{(j)} \\ \vdots \end{bmatrix}_{3\binom{n}{2}\times 3}}_{\text{svdCoeff}^\dagger} \mathbf{r}_\mathbf{v} - \underbrace{\begin{bmatrix} \vdots \\ \mathbf{t}^{(j)} - \mathbf{t}^{(i)} \\ \vdots \end{bmatrix}_{3\binom{n}{2}\times 1}}_{\text{svdConst}^\dagger} \right\|^2$$

where $M^{(i)} = \begin{bmatrix} R^{(i)} & \mathbf{t}^{(i)} \\ \mathbf{0} & 1 \end{bmatrix}$ is the (complete) bone matrix (after premultiplying the bind matrix and postmultiplying the inverse of it) the i-th bone, $w_\mathbf{v}^{(i)}$ is the skin weight of $\mathbf{v}$ to the i-th bone, and n is the number of bones to which $\mathbf{v}$ is bound with nonzero skin weight.

The computation is done at each frame using SVD († Refer to `src/App/Main:L349-475`). To reduce the number of computation, the result $\mathbf{r}_\mathbf{v}$, which depends on the set of bone indices i where $w_\mathbf{v}^{(i)}$ is nonzero, is cached into `centerOfRotationTable`, and used for other vertices. Finally, they are wrapped into an attibute and passed to the vertex shader. Then the final transform $\mathbf{v} \mapsto Q(\mathbf{v} - \mathbf{r}_\mathbf{v}) + \sum w_\mathbf{v}^{(i)} M^{(i)} \mathbf{r}_\mathbf{v}$ is constructed where Q is the QLERPed rotational matrix of $R^{(i)}$'s. Note for `vec3`s the transform becomes $\mathbf{n} \mapsto Q\mathbf{n}$.

Besides the center of rotation attribute, (the rotational parts of) bone matrices in quaternion form are also given as a uniform variable, packed in a `THREE.DataTexture` (`src/App/Main:L324-345`).

```
// src/Shader/SphericalBlendSkinning/Chunk/spherical_blend_skinnormal.chunk.vert
vec4 qlerped = getQLERP( skinWeight, boneQuatX, boneQuatY, boneQuatZ, boneQuatW );
mat3 skinMatrix3 = quatToMat3( qlerped );
objectNormal = skinMatrix3 * objectNormal;
#ifdef USE_TANGENT
  objectTangent = skinMatrix3 * objectTangent;
#endif

// src/Shader/SphericalBlendSkinning/Chunk/spherical_blend_skinning.chunk.vert
vec3 transl = vec3( 0.0 );
transl += vec3( boneMatX * vec4( centerOfRotation, 1.0 ) ) * skinWeight.x;
transl += vec3( boneMatY * vec4( centerOfRotation, 1.0 ) ) * skinWeight.y;
transl += vec3( boneMatZ * vec4( centerOfRotation, 1.0 ) ) * skinWeight.z;
transl += vec3( boneMatW * vec4( centerOfRotation, 1.0 ) ) * skinWeight.w;

transformed = skinMatrix3 * ( transformed - centerOfRotation ) + transl;
```

7 With Optimized Center of Rotations

The shader code is the same with SBS. The only difference is the content of the attribute `centerOfRotation`. It is first stored in `MainApp.optimizedCoRBuffers` as `Float32Array`s after fetching from the server, and passed into the shader if the skinning type is CoR. The advantage of this method is that there is no need to alter the vertex attribute at every render, since the center of rotations do not depend on bone matrices, but only on vertices (on rest pose), triangles, and skin weights, which are not changed over time. So, the attribute is set when the skinning type is changed (`src/App/Main:L273-278`).

The paper [LH16] introduces methods to optimize the baking time, however I did not implement those as it does not take that much time to execute. (To reproduce, click generate optimized CoR data button. Caution: the computer may become super laggy.) The code for generating optimized center of rotations is very straightforward to [LH16, Equation (4)]. See `cor/src/worker.ts`.

```
// cor/src/worker.ts
const approxOptimalCoRArray = _.range(vertices.length / 3)
  .map((vertexIndex) => {
    const wi = skinWeights[vertexIndex];
    const numerator = [0, 0, 0];
    let denominator = 0;

    _.chunk(triangleIndices, 3).forEach(([ia, ib, ic]) => {
      const va = getVertex(ia);
      const vb = getVertex(ib);
      const vc = getVertex(ic);
      const wa = skinWeights[ia];
      const wb = skinWeights[ib];
      const wc = skinWeights[ic];
      const wav = averageSkinWeight(wa, wb, wc);
      const vav = averageVertex(va, vb, vc);

      const area = getTriangleArea(va, vb, vc);
      const coeff = similarity(wi, wav) * area;
      numerator[0] += coeff * vav[0];
      numerator[1] += coeff * vav[1];
      numerator[2] += coeff * vav[2];
      denominator += coeff;
    });

    if (denominator === 0) {
      return [0, 0, 0] as Vertex;
```

```
  }

  return numerator.map((x) => x / denominator) as Vertex;
})
.flat();
```

8 Discussion

8.1 Performances

LBS and DQS both showed good performances without any lag on a consumer computer. CoR had a subtle delay probably because of passing large attributes, but it seems okay in general. However, SBS had a severe performance issue. It gets worse as it tries to do singular value decomposition in the browser at real-time, which is slower than native environments.

8.2 Artifacts

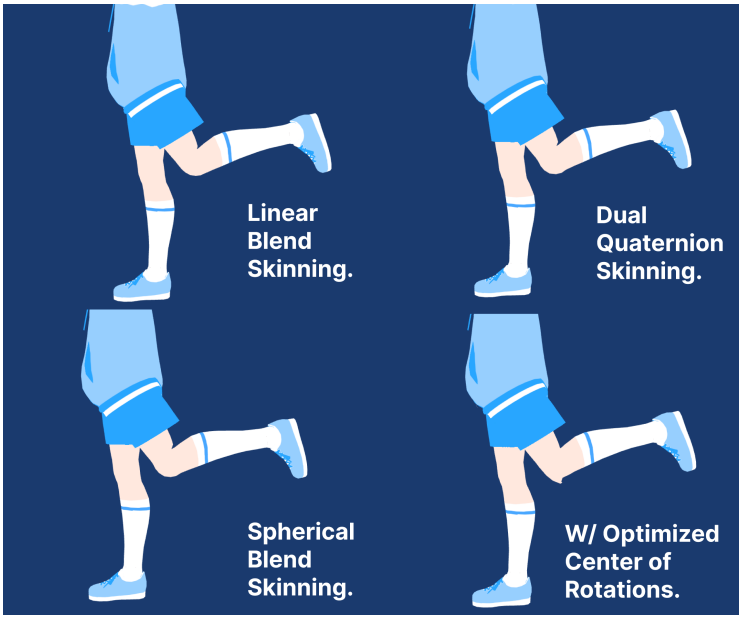

Figure 2: DQS and SBS have a crease and joint bulging effect, and CoR shows a discontinuous knee mesh.

Figures 1 and 2 show some artifacts of each method. First, the candy wrapper artifact is shown in LBS, when an arm is twisted (rotated about the arm direction.) And nonlinear methods, especially DQS ans SBS, show bulging artifacts. Moreover, as indicated in [LH16, Figure 4], with DQS or SBS, the knee part is bulged while there is a crease near the back of the knee which is clearer than LBS. However, a discontinuity is found with CoR, which is because the leg does not consist of a single mesh in my opinion. Since the optimized center of rotations is obtained for each mesh, regardless of neighbor meshes.

8.3 Unintended Bugs

Figure 3: An unintended bug producing weird results.

When running SBS and CoR at a high animation speed, the results seem very strange. And these artifacts disappear when one 'pauses' the animation by pressing Pause button. With some analyses, I think it is because the meshes and materials are separated and the app traverses the object sequentially. Especially, when each render step contains heavy operations such as large data copy or singular value decomposition, sequential update of attributes affect the actual results. It is probably the reason why the result is normal when the animation is paused by 'Pause' button (which internally keeps to render) but not when the execution is paused using browser debugger.

References

[KCŽO08] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics*, 27(4):1–23, October 2008.

[KŽ05] Ladislav Kavan and Jiří Žára. Spherical blend skinning. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games - SI3D '05*. ACM Press, 2005.

[LH16] Binh Huy Le and Jessica K. Hodgins. Real-time skeletal skinning with optimized centers of rotation. *ACM Transactions on Graphics*, 35(4):1–10, July 2016.