

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS no. 1538

Human Skeleton System Animation

Stephanie Cheng

Zagreb, June 2017

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1538

Animacija skeletnog modela čovjeka

Stephanie Cheng

Zagreb, lipanj 2017

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 6 March 2017

MASTER THESIS ASSIGNMENT No. 1538

Student: **Stephanie Cheng (0036473606)**
Study: Computing
Profile: Computer Science

Title: **Human Skeleton System Animation**

Description:

Investigate the model of human skeleton. Investigate direct and inverse kinematics especially in the context of human skeleton system. Develop an application for animation of human skeleton model. Implement a framework for analysis and comparison of the investigated models. Discuss the influence of parameters of the models on the output. Evaluate the results and the implemented algorithms. Implement the appropriate software solution. Use the C++ programming language and the OpenGL library. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

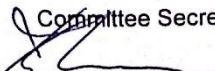
Issue date: 10 March 2017
Submission date: 29 June 2017

Mentor:



Full Professor Željka Mihajlović, PhD

Committee Secretary:



Assistant Professor Tomislav Hrkać, PhD

Committee Chair:



Full Professor Siniša Srbljić, PhD

Zagreb, 6. ožujka 2017.

DIPLOMSKI ZADATAK br. 1538

Pristupnik: **Stephanie Cheng (0036473606)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Animacija skeletnog modela čovjeka**

Opis zadatka:

Proučiti skeletni model čovjeka. Proučiti direktnu kinematiku i mogućnosti primjene na skeletnom modelu. Razraditi implementaciju ostvarivanja pokreta čovjeka temeljem proučenog modela. Načiniti programsku implementaciju koja omogućuje analizu ostvarenih rezultata. Na različitim primjerima prikazati ostvarene rezultate. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama.

Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje OpenGL. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 10. ožujka 2017.
Rok za predaju rada: 29. lipnja 2017.

Mentor:



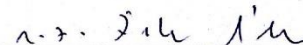
Prof. dr. sc. Željka Mihajlović

Djelovođa:



Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:



Prof. dr. sc. Siniša Srbljić

Table of Contents

1. Introduction.....	1
2. Skeletal animation theory	2
3. Used tools.....	5
3.1. OpenGL.....	5
3.1.1 Libraries.....	8
3.2. Assimp.....	8
3.2.1. Assimp Data Structure	8
3.3. Blender	10
3.4. Library Linmath	11
4. Implementation.....	12
4.1. Classes.....	12
4.2. Basics of OpenGL.....	14
4.3. Shaders.....	15
4.4. Importing through Assimp	16
4.5. Assigning bones to vertices.....	17
4.6. Drawing the skeleton	18
4.7. The leaf bone problem.....	21
4.8. Animating and interpolation	25
4.8.1 Interpolation	27
4.9. Adding masks	29
4.10. Blending animations.....	29
4.10.1. Animation Blend.....	30
4.10.2. Transitions.....	31
5. Application Use.....	33
5.1. Foreword on importing the model	33
5.2. Application Features	33
5.3. Application manual	36
6. Conclusion	37
7. BIBLIOGRAPHY	38

1. Introduction

Skeletal animation is a technique in computer animation in which a character or other articulated object is represented in two parts: a surface representation used to draw the character called skin or mesh and a hierarchical set of interconnected bones called the skeleton or rig used to animate (pose and keyframe) the mesh [1].

Skeletal animation has become the industry standard way for animating organic models such as characters and nonorganic 3d models.

The technique was introduced in 1988 by Nadia Magnenat Thalmann, Richard Laperrière, and Daniel Thalmann [ref]. This technique is used in virtually all animation systems where simplified user interfaces allows animators to control often complex algorithms and a huge amount of geometry; most notably through inverse kinematics and other "goal-oriented" techniques. In principle, however, the intention of the technique is never to imitate real anatomy or physical processes, but only to control the deformation of the mesh data [1].

This project explores building a system that reproduces animations based on a skeleton system, allows masking and blending of animations, and displaying the skeleton. It has been built from the ground up using OpenGL to render the scene.

This paper will explain the features and techniques used in our application, the various libraries, animation theory behind the scenes.

2. Skeletal animation theory

Skeletal Animation is a technique in computer animation in which an animated object represented by a mesh is animated using a hierarchical structure of bones called a **Skeleton** or **Rig**.

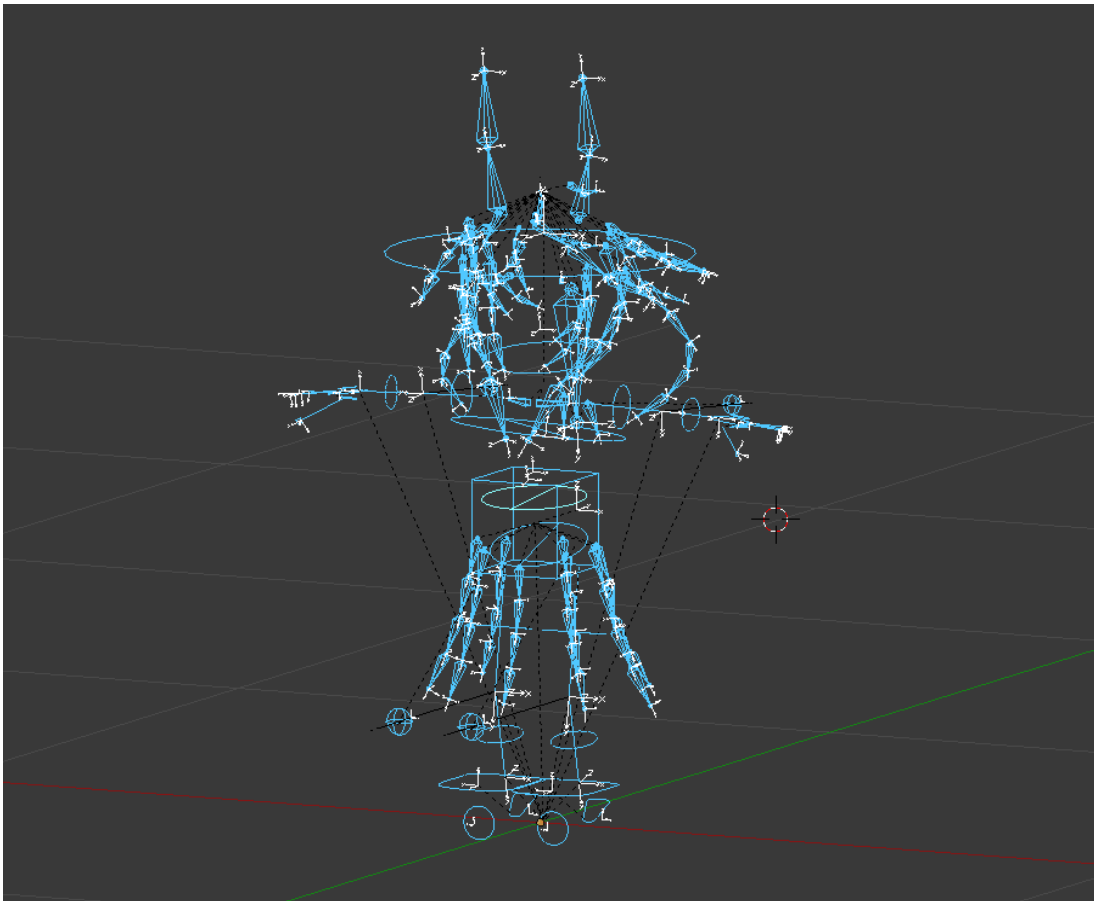


Figure 1 A completed Skeleton

Rigging is a process of building the skeleton which will drive the animation for a chosen mesh or meshes, thus making the mesh able to move (Figure 1). It includes linking the bones in a hierarchy, setting constraints on the bones' movement, and setting up controls which are aids for the animator.

The bones are organized in a hierarchical manner, this means the bones have child/parent relationships, every bone except the root has one parent. When a bone moves it also moves all its children, but not the parent. This means that when calculating transformations of a bone, we need to combine the transformations of all the parent bones up to the root [2].

Even though most SDKs and game engines define a skeleton as having bones, that's incorrect. It has **joints**. The bones are the implied connections between joints.

A bone is usually visualized by its head and tail, the head is in place of the joint it represents, transforming the bone does not move the head, while the tail represents the next joint this bone is connected to or if it is a leaf bone it does not represent anything and essentially does not exist. That's why most file formats for storing skeletal and animation data do not store the placement of the tail. However, many rigging programs use this kind of visualization since it is easy to perceive and control the transformation using the tail.

Skinning is the process of attaching vertices to the bones, an artist does this using a process called weight painting, where for a select bone the artist assigns weights to the mesh ranging from 0.0 (no influence) to 1.0 (full influence). A vertex can be influenced by multiple bones with different weights, the total sum of weights on a vertex should be 1, this is handled by the modeling program (Figure 2).

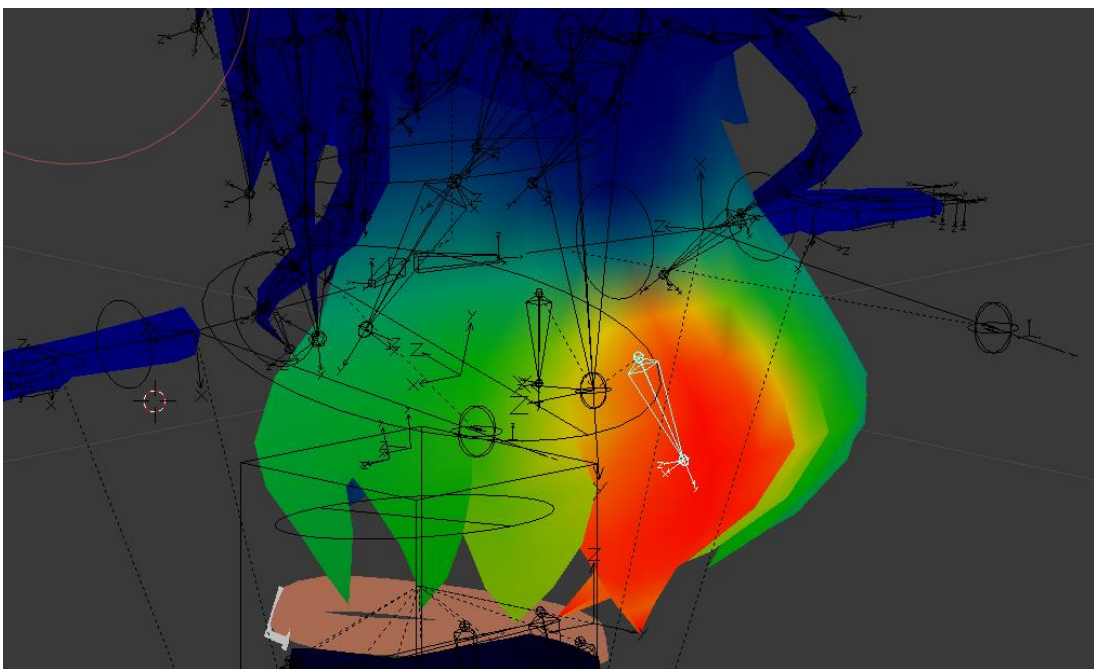


Figure 2 Skinning a mesh to the skeleton, the colors indicate weight values, with red being 1 (full influence) and blue being 0 (no influence)

Once the rigging and skinning procedures are complete, the skeleton can be posed to deform the mesh, bone movement influences assigned vertices multiplied by weight.

The Bind pose of the skeleton is the transform (position, rotation and scale) of the bones in which no deformation occurs on the mesh. Movement from the bind pose deforms the mesh.

To animate a vertex, we need to transform it with the bone transformation. Each bone has its local space called bone space, the transformation for each bone is in that space, to transform the vertex we first move the vertex into bone space, then we transform it with the transformation obtained by combining the transformations of all the parent of the bone and the bone itself.

The approach we take when building the application is to first calculate the final transform matrix of a bone and pass it into the shader that will then transform the vertex. The final transform matrix is applied to the vertex with a weight of influence that bone has on the vertex.

A **keyframe** in 3d animation is a location on an animation timeline that stores transformation information such as scale, rotation and position of that point in time of the animation. It can also store other information that is animated in time, such as slider values on various modifiers, but in this project, we are working with bone keyframes only.

When creating an animation in an animation software the artist places the bones in desired position at a desired time and marks a keyframe on those bones. When playing back the animation, the keyframes are interpolated in various ways to play a smooth transition between the transformations of the bone.

A 3d **animation** consists of a timeline populated with keyframes of various bones. An animation consists of channels; a **channel** is a keyframe timeline of only one bone.

Framerate is the number of drawn frames per second, however animations have their own predefined framerate with which they define their timeline, rather than defining it by fixed time intervals. So, when playing an animation, we need to know its framerate to determine its duration or the time at which each keyframe is defined.

Framerate of the animation is independent of the framerate at which an application is running, and the animation will be the same length no matter which framerate the application is running at.

Animation **masking** is the process of playing a different animation on the masked part of the skeleton than the base animation being played. Such as playing a walking animation, but masking a look around animation on the head of the character.

Blending animation is the process of combining two or more animations by interpolating their transformation by a specified weight. The result is a mix of the two, most commonly used to make smooth transitions from one animation to another, or to make a new in between animation, such as making the character turn in angles in between 0 and 45 degrees.

3. Used tools

3.1. OpenGL

Open Graphics Library (OpenGL) is mainly considered an API (an Application Programming Interface) that provides us with a large set of functions that we can use to manipulate 2D and 3D vector graphics and images. However, OpenGL by itself is not an API, but merely a specification, developed and maintained by the Khronos Group [3].

The OpenGL specification specifies exactly what the result/output of each function should be and how it should perform. It is then up to the developers implementing this specification to come up with a solution of how this function should operate [3].

OpenGL is by itself a large state machine: a collection of variables that define how OpenGL should currently operate. The state of OpenGL is commonly referred to as the OpenGL context. When using OpenGL, we often change its state by setting some options, manipulating some buffers and then render using the current context [3].

OpenGL works in 3D space, but to represent that 3D space on the screen a large portion of OpenGL is transforming 3D space coordinates into pixels on the screen. This process is managed by the **graphic pipeline** of OpenGL (Figure 3). The graphics pipeline can be divided into two large parts: the first transforms your 3D coordinates into 2D coordinates and the second part transforms the 2D coordinates into actual colored pixels [4].

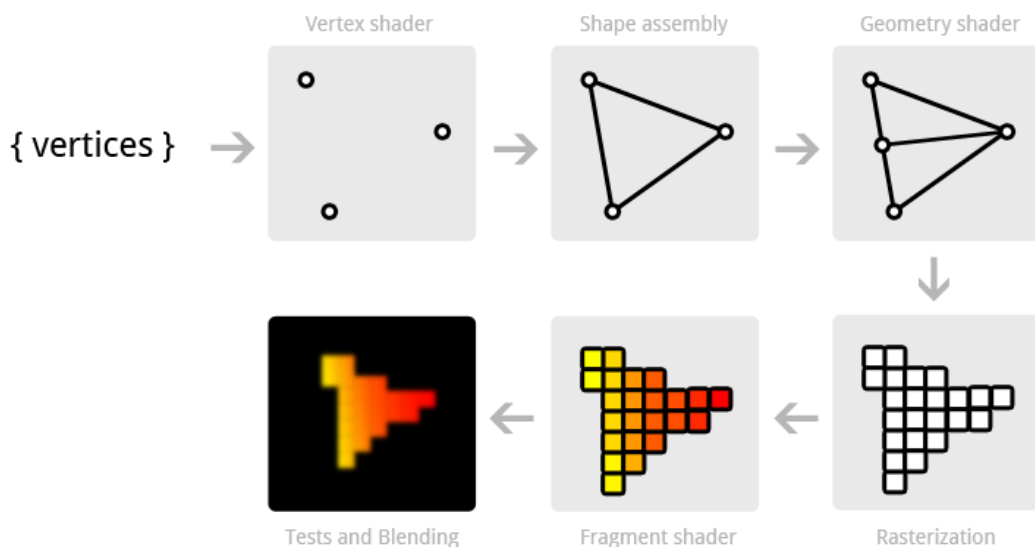


Figure 3 OpenGL graphic pipeline [5]

The programmable parts of the graphic pipeline are the vertex shader and the fragment shader, less commonly geometry shader and more recently tessellation.

The **vertex shader** inputs are vertices and their attributes, and is used to output other attributes and finalize the positions of the vertices.

The primitive assembly stage takes as input all the from the vertex shader that form a primitive and assembles all the point(s) in the primitive shape given [4]. Primitives are the type of render data we want to represent with vertices such as points, lines, and most commonly triangles.

The output of the primitive assembly stage is passed to the geometry shader. The geometry shader takes as input a collection of vertices that form a primitive and can generate other shapes by emitting new vertices to form new (or other) primitive(s).

The output of the geometry shader is then passed on to the rasterization stage where it maps the resulting primitive(s) to the corresponding pixels on the final screen, resulting in fragments for the fragment shader to use [4].

A **fragment** in OpenGL is all the data required for OpenGL to render a single pixel. The main purpose of the **fragment shader** is to calculate the final color of a pixel and this is usually the stage where all the advanced OpenGL effects occur. Usually the fragment shader contains data about the 3D scene that it can use to calculate the final pixel color like lights, shadows, color of the light and so on. The last stage is the tests and blending stage, here fragments are mixed based on their alpha, or discarded if they are not visible [4].

Defining the vertex shader and the fragment shader is required, while the geometry shader is optional, in our application we leave the geometry shader as default, while we provide two vertex and fragment shaders, for rendering models and rendering the models skeleton.

The language used in vertex and fragment shader is GLSL, GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.

Once the vertices pass through the vertex shader they should be in **normalized device coordinates** (NDC). That is, the x and y coordinates of each vertex should be between -1.0 and 1.0 and between 0.0 and 1.0 in z (a 2x2x1 cuboid), all vertices outside of that range won't be visible. The users view is at the origin, and is looking at +z.

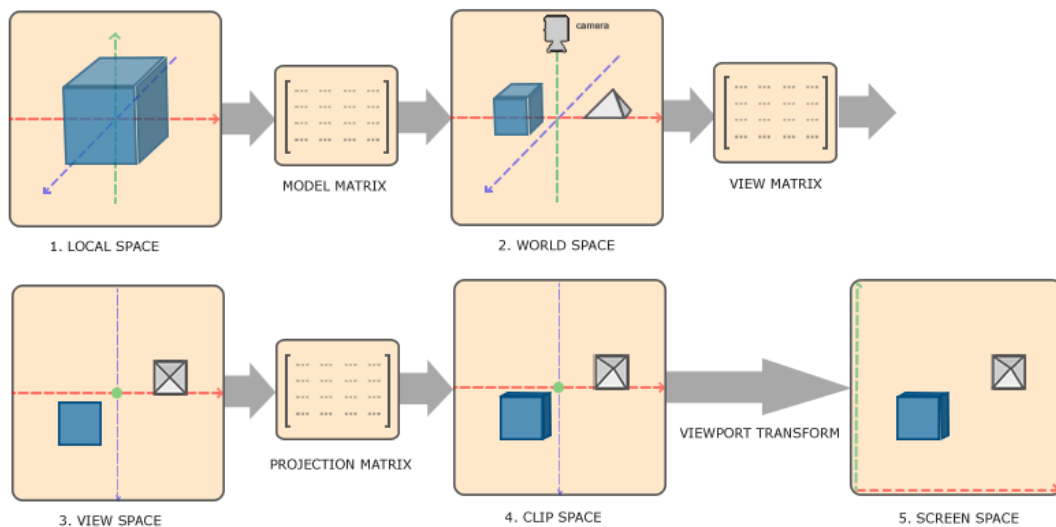


Figure 4 Transformations between spaces in OpenGL [6]

Transforming coordinates to NDC and then to screen coordinates is usually accomplished in a step-by-step fashion where we transform an object's vertices to several coordinate systems before finally transforming them to screen coordinates (Figure 4). The advantage of transforming them to several intermediate coordinate systems is that some operations/calculations are easier in certain coordinate systems as will soon become apparent. There is a total of 5 different coordinate systems that are of importance to us [6]:

- Local space (or Object space)
- World space
- View space (or Eye space, Camera Space)
- Clip space (NDC)
- Screen space

Model matrix represents the placement of the object in the world, the view matrix the placement of the camera, while the projection matrix is used to project the objects onto the camera lens in various ways, most notably orthographic projection and perspective projection.

Orthographic projection is a form of parallel projection, in which all the projection lines are orthogonal to the projection plane.

Perspective projection is more natural to our perception of the world, with distant things appearing smaller than closer things, this is the projection we use in our application.

Old OpenGL conventions state that the camera space is right handed, the camera is positioned at the origin and is looking at the negative z axis, while in NDC space it is looking at the positive z axis. However, in modern OpenGL we can use arbitrary matrices, and are not forced to define our eye or object spaces right-handed, it used to be something akin to a default convention. A lot of libraries still follow this convention, including the ones we use.

So, our camera space is defined as right handed with the camera looking at the negative z axis, with its up axis being y and right axis being x. Moving the camera around is achieved by instead moving the scene while the camera stays fixed; we are moving the entire scene inverse of what would be the movement of the camera.

3.1.1 Libraries

Other libraries used for OpenGL include:

opengl32 included with Microsoft SDK

GLFW a simple API for creating windows, contexts, surfaces and receiving inputs and events

GLEW an OpenGL Extension Wrangler Library that provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform

SOIL (Simple OpenGL Image Library) a tiny C library used primarily for uploading textures into OpenGL.

3.2. Assimp

Open Asset Import Library (Assimp) is a portable Open Source library to import various well-known 3D model formats in a uniform manner. It is tailored at typical game scenarios by supporting a node hierarchy, static or skinned meshes, materials, bone animations and potential texture data.

Written in C++, it is available under a liberal BSD license. Assimp loads all input model formats into one straightforward data structure for further processing.

3.2.1. Assimp Data Structure

The assimp library returns the imported data in a collection of structures.

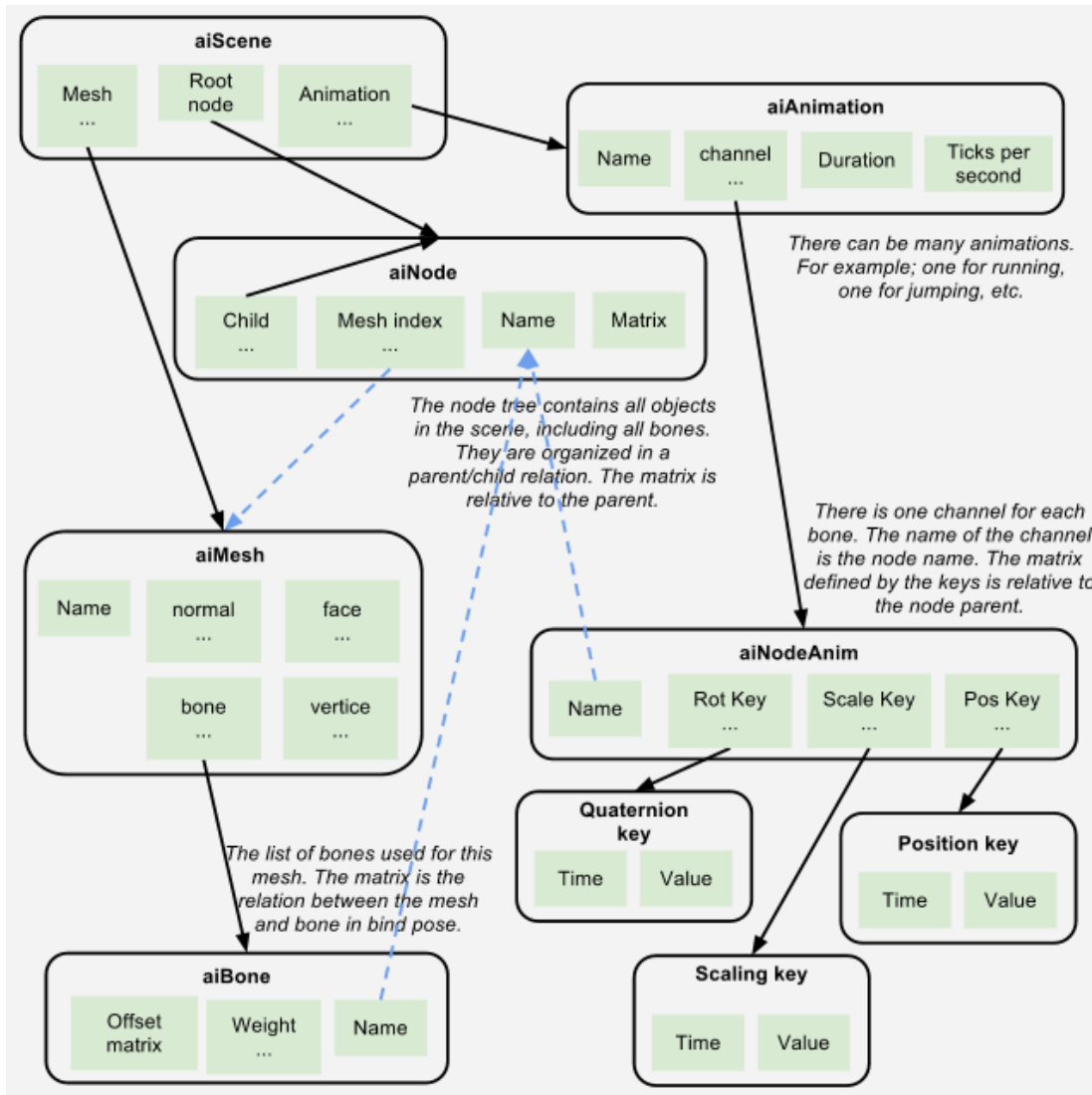


Figure 5 Assimps imported structure [7]

aiScene forms the root of the data, from here you gain access to all the nodes, meshes, materials, animations or textures including the number of each these structures that were read from the imported file (Figure 5). The aiScene is returned from a successful call to Assimp::Importer::ReadFile(), ailmporFile() or ailmporFileEx() [8].

aiNodes are little named entities in the scene that have a place and orientation relative to their parents. Starting from the scene's root node all nodes can have 0 to x child nodes, thus forming a hierarchy [8].

An aiNode can represent a mesh or multiple meshes, a bone or an animation channel, the specifications of which are stored in other nodes that can be referred to by name shared with aiNode. Each aiNode stores their transformation matrix which defines the nodes coordinate system relative to its parent. A pointer to its children, its index and its Name.

An aiMesh houses a single or multiple meshes, the vertices, faces, normals and uv coordinates of said meshes, material data (not pictured), and has a list of bones that influence that mesh. If only a part of the skeleton influences the mesh only those bones will be contained in the mesh. This in turn means that if a bone does not influence anything (like a control bone) it will not be referenced by any mesh, it is still possible to use this bone as it is stored as an aiNode only.

aiBone represents one bone in the skeleton of the mesh. Each bone has a name by which it can be found in the node hierarchy, an array of vertex weights and a 4x4 offset matrix transforming from mesh space to bone space, also called the inverse bind pose transform in Assimp [7]. The reason why we need this matrix is because the vertices are stored in the usual local space. This means that even without skeletal animation support our existing code base can load the model and render it correctly. But the bone transformations in the hierarchy work in a bone space and every bone has its own space which is why we need to multiply the transformations together. So, the job of the offset matrix is to move the vertex position from the local space of the mesh into the bone space of that bone [2].

aiAnimation node is a single named animation. An animation in this context is a set of keyframe sequences where each sequence describes the orientation of a single node in the hierarchy over a limited time span. An aiAnimaiton contains aiNodeAnims which represent a channel of animation. A channel is a set of keyframes over a period that influence a single Bone. aiNodeAnims store a bones rotation, position and scale keyframes and their count.

aiNodes are stored in a hierarchy, while other node types are stored in a list, to access the full information of a given node we must find the corresponding structure by name.

In the case of extracting skeleton and animation data, the skeleton is a hierarchical structure and aiNodes corresponding to bones follow this defined structure, so we must traverse the aiNode hierarchy to properly propagate parental transform matrices, determine if an aiNode is a bone and then refer to the aiBone node by name stored in the aiNode.

Matrices stored in the assimp node structures are row major type, linmath and OpenGL both use column major type matrices, so when extracting matrices from the assimp structure it is important to transpose them before storing.

3.3. Blender

Blender is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation.

Blender was used to model, rig, texture, animate and export various 3d models for testing this application (Figure 6).

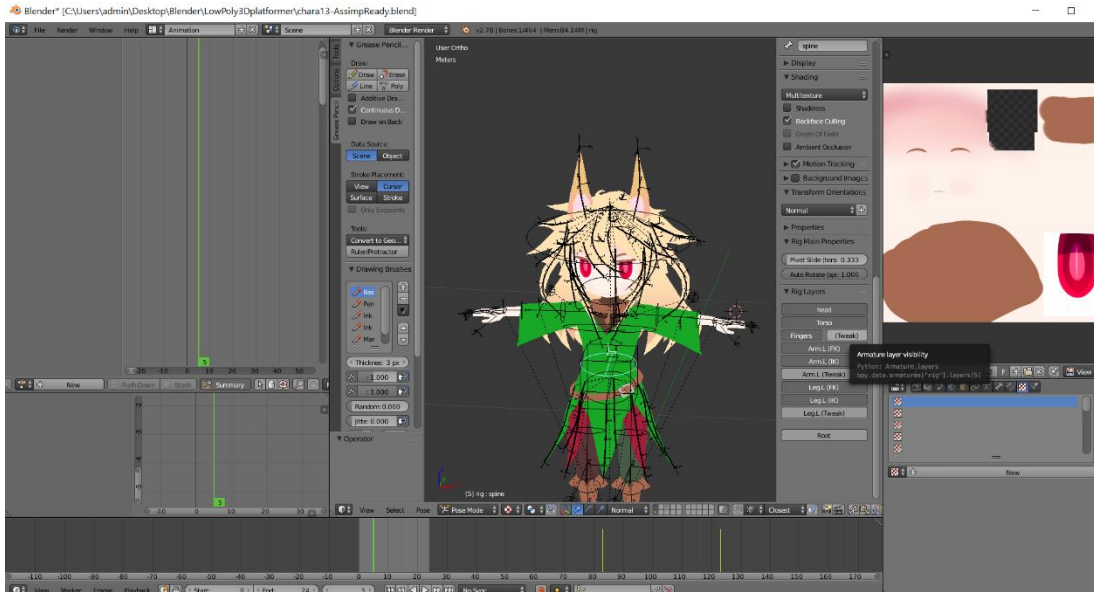


Figure 6 Blenders user interface for animation

3.4. Library Linmath

Library Linmath.h is a free small library for linear math as required for computer graphics, it supports the most used types of data structures required for programming graphics including vector of 3 elements, vectors of 4 elements, quaternions and 4 by 4 matrices in column major order.

4. Implementation

4.1. Classes

The application that is the subject of this paper is developed in C++, and adopts an object-oriented approach to its structure. As such, it uses classes to organize code.

The Camera class handles the logic that controls the users point of view. The cameras projection type is perspective to give the impression of depth and a more realistic depiction of 3d space. The camera class is used to move the camera around, calculate needed vectors such as the forward up and right vectors, calculate look at and the view and projection matrices, and set field of view and clipping planes.

The Shader class is the owner of the shader program, the application uses two shaders one for displaying the loaded object and another to display the skeleton structure of that object. The Shader class handles reading, compiling shaders, checking for errors and creating the shader program.

Model class represents a single model imported with assimp, that model can consist of multiple meshes, materials, textures and a skeleton, that skeletons animations and channels, user created masks, blends and transitions. It handles loading the model, animating the model, animation controls, position scale and rotation of itself, drawing the model on the screen.

It contains the following structs

```
typedef struct BoneInfo {
    mat4x4 boneOffset;
    mat4x4 finalTransformation;
};
```

Vector of BoneInfos keeps a bones boneOffset matrix that transforms vertices to bone space and its finalTransformation matrix which is the matrix fed into the OpenGL pipeline to determine the final positions of vertices.

```
typedef struct AnimationInfo {
    Time::time_point tStart;
    Time::time_point tNow;
    int animation;
    float animationTime;
};
```

AnimationInfo stores data regarding a played animation, when did it start playing, current time, what animation and duration.

```
typedef struct TransitionInfo {
    int anim1;
    int anim2;
    int frameStart;
    int frameEnd;
    Time::time_point tAnimationStart;
    Time::time_point tAnimationCurrent;
};
```

TransitionInfo stores user created transitions, when creating a transition we need the indexes of the two animations being blended, frame in which the transition starts and ends, and the current and start time of playback.

```
typedef struct BlendInfo {
    int anim1;
    int anim2;
    float weight;
    Time::time_point tAnimationStart;
    Time::time_point tAnimationCurrent;
};
```

BlendInfo stores user created blends which need animations that are being blended, and the weight at which we blend them (0 being 100% of anim1 and 1 being 100% of anim2).

Mesh class represents a collection of polygons and the vertices, coordinates and normal of those polygons. It handles setting up OpenGL buffers. Assigning bones to vertices and drawing uniforms.

```
typedef struct Vertex {
    vec3 position;
    vec3 normal;
    vec2 texcoord;
};
```

A single vertex has information on its position, normal and uv coordinates.

```
struct MatStruct {
    mat4x4 mat;
};
```

Used as a workaround to tightly pack matrices to send them easily to the shader.

Linmaths mat4x4 is an array, so vector<mat4x4> is a vector of pointers that point to their array values, thus making the matrices scattered. A struct is always tightly packed, so putting the array in a struct and then a vector<MatStruct> makes sure all the matrices are tightly packed.

```
typedef struct VertexBoneData {
    GLuint boneID[NUM_BONES_PER_VERTEX];
    GLfloat weights[NUM_BONES_PER_VERTEX];
};
```

Vertex Bone data keeps bone weights and boneID information of a vertex.

NUM_BONES_PER_VERTEX defines how many bones can influence a single vertex, it varies between different applications, a common number is usually 3 or 4, this application supports up to 4 bones.

SkeletonMesh class is similar to the mesh class, but without most of the functions of the mesh class. Each Model has exactly one or no SkeletonMesh. The skeleton mesh stores vertices in proper positions that represent placements of bones. It also handles drawing the uniforms and assigning those vertices to appropriate bones.

The Texture class handles loading a texture with soil, assigning an OpenGL id to the texture, binding the texture, and setting up OpenGL parameters for handling the texture.

4.2. Basics of OpenGL

OpenGL uses buffers to store large amounts of data that is passed to the graphic card.

For sending vertices we use the vertex buffer objects (VBO) that can store a large number of vertices in the GPU's memory. Once we load the vertices into our program we must send them to the graphic card. We first must create and fill the VBO for each vertex attribute such as position, normal and uv coordinate. And indicate which vertex buffer corresponds to which attribute.

Vertex array objects (VAO) save information about which Vertex Buffer Object is connected to which attribute of the vertex shader, so when drawing objects, we simply bind the VAO.

Element buffer objects (EBO) is used to store indices of vertices, so even though a vertex is shared between triangles it is stored only once, while the EBO takes care that both triangles use it. This drastically reduces the memory needed to store meshes.

This is handled by the mesh class that loads vertex positions, normals, uv coordinates, bone weights and bone ids.

Since we want OpenGL to keep drawing images on the screen we use the render loop located in the main function, the render loop keeps running and drawing images until we tell OpenGL to stop.

Inside the render loop we handle all the changes that can happen to an OpenGL scene. Here is the code that's executed after we load the model including calculating all necessary transformations to handle animation moving the camera.

In the render loop, we send the needed uniform to the shaders including the model matrix, the view matrix, the projection matrix, the bone transformations and other. We also handle drawing objects, handling key inputs, swapping buffers and clearing buffers.

The Draw function of the mesh handles binding the textures and the necessary buffers, and drawing into the buffer with the passed shader.

4.3. Shaders

We use two shaders in this application, a general shader to shade models and animate them, and a second shader that renders the skeleton.

To animate the vertices, we take the transformations of the bones, combine them by weight and multiply with the position of the vertex.

```
mat4 bonetransform = gBones[boneIDs[0]] * weights[0];
bonetransform += gBones[boneIDs[1]] * weights[1];
bonetransform += gBones[boneIDs[2]] * weights[2];
bonetransform += gBones[boneIDs[3]] * weights[3];

newpos = bonetransform * vec4(position, 1.0f);

gl_Position = projection * view * model * newpos;
```

The gBones is the uniform storing the bone transformation while boneIDs store a maximum of 4 bone IDs that influence the current vertex.

The second shader used for rendering the skeleton operates much in the same way, but with only one bone influencing a vertex.

4.4. Importing through Assimp

Before we start rendering our scene we need to load chosen objects into it, this is done using Assimp. Once Assimp has successfully loaded the file it will return the reference to the root of its data structure in `aiScene`, `aiScene` objects are owned by Assimp, and not the caller, to make sure the loaded data does not get destroyed we must store a pointer to the data.

From `aiScene` we must extract every mesh, the meshes vertices, materials of the mesh, assign bones to vertices of the mesh and the skeleton information of the model.

A vertex is described by its position, normal, and uv coordinates, this vertex data is stored in a struct of the same name. Once we extract all vertices of a mesh, for OpenGL to form triangles from them we must store a vector of indices that represent vertices, bunches of three indices starting from the first forms one triangle, many vertices are shared between triangles so this type of storage greatly reduces needed space.

From materials, we only use the diffuse texture. For every texture, instantiate a `Texture` class, load the texture using SOIL and generate an OpenGL texture ID and set up OpenGL settings. After needed data is extracted we instantiate a `Mesh` class, and setup that mesh and its OpenGL buffers with the extracted data.

```
glGenBuffers(1, &ebo);
glGenVertexArrays(1, &vao);
glGenBuffers(1, &vbo);

glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0],
GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint),
&indices[0], GL_STATIC_DRAW);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)0);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, normal));

glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, texcoord));

glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);
```

The function `glVertexAttribPointer` defines an array of generic vertex attribute data to be sent to the vertex shader. At location 0 we send the positions, at location 1 we send the normal, and at location 2 we send the uv coordinates.

Next we must assign the bones to the vertices of the mesh detailed below.

4.5. Assigning bones to vertices

The last step of loading mesh data is to assign each vertex of the mesh its bones and their weights. During this procedure, we also populate the various structures that hold bone information and animation information. We iterate through all the bones contained in `aiMesh`, and for each of them save appropriate information in the maps.

`std::map<std::string, int> boneMap` maps the name of the bone to its index, this is needed since we identify bones by their name. If a bone wasn't previously saved in the map, we create its bone info and populate it with necessary data.

`std::map<int, BoneInfo> boneInfoMap` is a map with the index of the bone being the key and the `BoneInfo` struct which holds bone data being the value.

We then need to assign the weights of that bone to all the vertices of the mesh it affects.

```
for (int j = 0; j < mesh->mBones[i]->mNumWeights; j++) {
    int vertexID = mesh->mBones[i]->mWeights[j].mVertexId;
    float Weight = mesh->mBones[i]->mWeights[j].mWeight;

    if (Weight <= 0.1) continue;

    //std::cout << "bone assigned weight " << Weight << std::endl;
    createdMesh->AddBoneData(boneIndex, Weight, vertexID);
}
```

A `vertexID` is the index at which the vertex is stored in the list of vertices, this index is unique between the vertices of a mesh and can be used as its identifier.

Each mesh has a `std::vector<VertexBoneData> vbd` that keeps `VertexBoneData` data for each vertex

```
void Mesh::AddBoneData(GLuint boneID, float weight, GLuint vertexID)
{
    if(vbd.size() == 0) vbd.resize(vertices.size());

    for (GLuint i = 0; i < NUM_BONES_PER_VERTEX; i++) {
        if (vbd[vertexID].weights[i] == 0.0f) {
            vbd[vertexID].boneID[i] = boneID;
            vbd[vertexID].weights[i] = weight;
            return;
        }
    }
}
```

For each vertex the bone affects, we save the information as follows:

We iterate over the weights of the vertex, if we find 0 that means it is an unused slot since nothing is affecting the vertex. We save the new weight and the BoneID of the bone that's influencing the vertex. It is possible that more than 4 bones are affecting a vertex, in that case we simply ignore the all but the first 4 bones.

Finally, we need to save the animation keyframes for the bone.

```
for (int anim = 0; anim < scene->mNumAnimations; ++anim) {
    const aiAnimation* pAnimation = scene->mAnimations[anim];
    std::string animName = pAnimation->mName.data;
    for (int i = 0; i < pAnimation->mNumChannels; i++) {
        //save each channell to a bone
        if (pAnimation->mChannels[i]->mNodeName.data == boneName) {
            animations[animName][boneName] = pAnimation->mChannels[i];
            break;
        }
    }
}
```

After we assign the bones to the vertices we need to send that info to the OpenGL pipeline.

```
glBindVertexArray(vao);

glGenBuffers(1, &boneVB);
glBindBuffer(GL_ARRAY_BUFFER, boneVB);
glBufferData(GL_ARRAY_BUFFER, sizeof(vbd[0]) * vbd.size(), &vbd[0],
GL_STATIC_DRAW);
//bone IDs
glEnableVertexAttribArray(3);
glVertexAttribIPointer(3, 4, GL_INT, sizeof(VertexBoneData), (const
GLvoid*)0);
//weights
glEnableVertexAttribArray(4);
glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, sizeof(VertexBoneData),
(const GLvoid*)16);

glBindVertexArray(0);
```

4.6. Drawing the skeleton

After loading each mesh, we have to construct the skeleton, this action is solely for the purpose of displaying that skeleton in the application and is not necessary to make the animation work.

```

void Model::GetSkeletonBonesRec(aiNode* pNode, mat4x4 parentTransform, vec3
startPosition, int prevID, bool recThroughSkeleton)
{
    //bones are joints, we draw lines that connect them - we need previous
position and this position
    std::string NodeName (pNode->mName.data);

    mat4x4 nodeTransformation;
    aiMatTolinMat (nodeTransformation, pNode->mTransformation);
    //pAnimation->mChannels[0]->mNodeName

    //mat4x4 res1;
    mat4x4 res;
    vec3 pos;
    pos[0] = startPosition[0];
    pos[1] = startPosition[1];
    pos[2] = startPosition[2];

    mat4x4_mul(res, parentTransform, nodeTransformation);

    int ID = -1;

    std::string cName = NodeName.substr(NodeName.find_last_of("_") + 1);

    if (boneMap.find(NodeName) != boneMap.end() || cName == "end") {
        if (cName == "end") {
            ID = prevID;
        }
        else {
            ID = boneMap[NodeName];
        }
        recThroughSkeleton = true;

        pos[0] = res[3][0];
        pos[1] = res[3][1];
        pos[2] = res[3][2];

        skeletonMesh.AddBoneData (prevID, ID, startPosition, pos);
    }

    if (recThroughSkeleton) {
        for (GLuint i = 0; i < pNode->mNumChildren; i++) {
            GetSkeletonBonesRec (pNode->mChildren[i], res, pos, ID, rec-
ThroughSkeleton);
        }
    }
    else {
        mat4x4 I;
        mat4x4_identity(I);
        for (GLuint i = 0; i < pNode->mNumChildren; i++) {
            GetSkeletonBonesRec (pNode->mChildren[i], I, pos, ID, rec-
ThroughSkeleton);
        }
    }
}

```

The GetSkeletonBonesRec is a recursive function that traverses the aiNode hierarchy. Once we find the root of the skeleton we start passing down the parentTransform pass (the roots

parentTransform is an identity matrix). At each bone node, we calculate the transformation matrix, and extract the position which will be sent to the AddBoneData function.

We build the skeleton by manually placing vertices where the location of the joints should be, we assign weights according to which bone they represent and we draw lines in between them to represent bones.

We achieve this by using OpenGLs draw lines function. OpenGL draws lines by taking a vector of positions and grouping them two by two, so for each line we draw we must provide two positions, the head of the bone and the tail of the bone, which are actually joints of the skeleton.

With this we are creating a bit of redundancy, needing to store a joint position twice, once as a head and once as a tail. However, this is an easy and effective method of drawing the skeleton.

```
void SkeletonMesh::AddBoneData(GLint prevBoneID, GLuint BoneID, vec3 pos,
vec3 pos2)
{
    //bone head belongs to the previous bones transforms, tail belongs to
current
    VertexBoneData bd;
    bd.boneID[0] = prevBoneID;
    bd.weights[0] = 1.0f;
    //if this is the root, it has no previous bone
    if (prevBoneID == -1) {
        //bd.weights[0] = 0.0f;
        bd.boneID[0] = BoneID;
    }
    bones.push_back(bd);

    //add the position
    positions.push_back(pos[0]);
    positions.push_back(pos[1]);
    positions.push_back(pos[2]);

    VertexBoneData bd2;
    bd2.boneID[0] = BoneID;
    bd2.weights[0] = 1.0f;

    bones.push_back(bd2);

    //should aslo add end position
    //temp end position is pos + 1.0y

    positions.push_back(pos2[0]);
    positions.push_back(pos2[1]);
    positions.push_back(pos2[2]);

    //index of poisitions and index of bones is the vertexID
    return;
}
```

The SkeletonMeshs' AddBoneData function adds a single bone to the vector of bones. The head position is the position of the previous bone's tail, and the tail position is the position of the current joint, calculated by multiplying the parent transform matrix and the node matrix, and extracting the position from the result.

The head of the bone is not controlled by the current bone, but by the previous one (as it is its tail), that's why we assign it to the ID of the previous bone, while the tail is assigned to this bone. All weights are 1 as joints belong to one bone only.

Sending the data to OpenGL buffers is done similarly to sending the data for a normal mesh, but without normal and uv coordinate info.

The edge cases are the first bone aka the root bone, that does not have a previous bone, and the leaf bones that do not have tail information. The root bone will take ownership of both its tail and head. In the case of leaf bones, the problem is a bit bigger, explained in the following subchapter.

4.7. The leaf bone problem

We previously mentioned that most engines define a skeleton through bones, but that is incorrect, it is defined through joints. The bones are implied connection between joints. When looking at it that way we realize there's no bone extending pass the leaf joint, aka leaf bones have no length. And yet some programs do display the leaf bone (Figure 11) with a length and orientation.

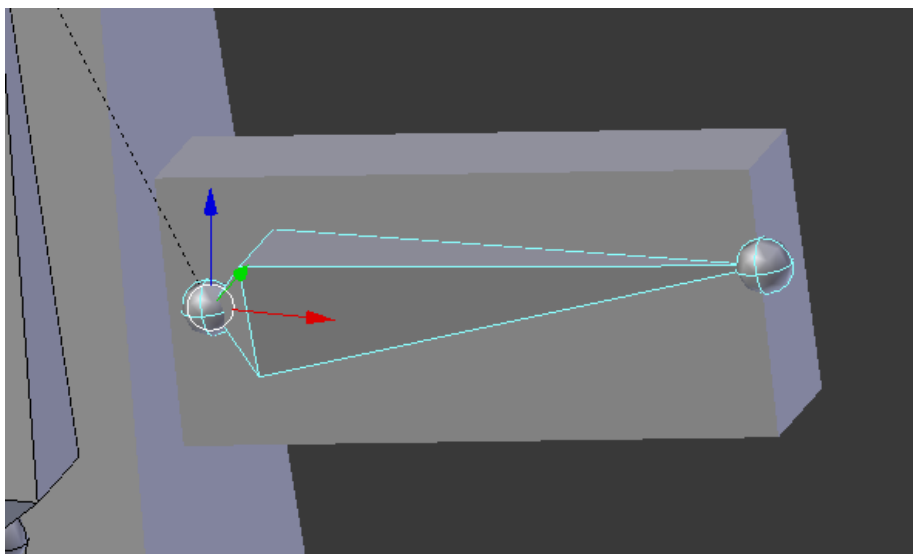


Figure 7 A leaf bone, the head (left sphere) is the actual joint position, while the tail (right sphere) visualizes movement, but is not necessary, and its information is not saved in most file formats.

When importing through Assimp there is no data that specifies length of the bones, we simply draw bones as connections between joints, this is why in our application and in many others bones that are offset (Figure 12) (aka appear disconnected from the parent) actually are displayed with a bone connecting them (Figure 13).

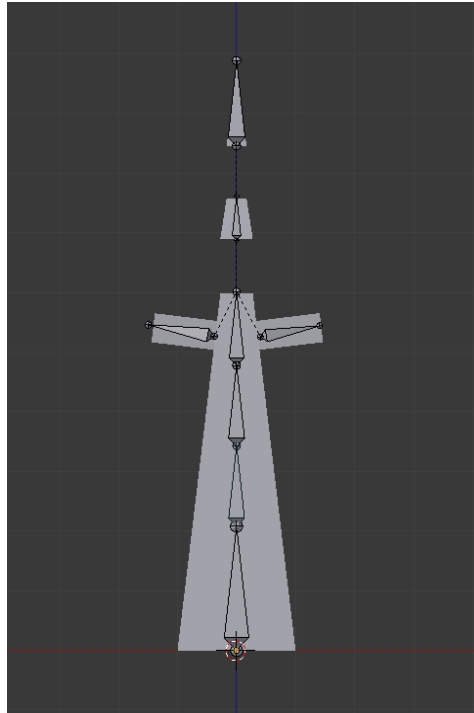


Figure 8 A rig displayed in blender

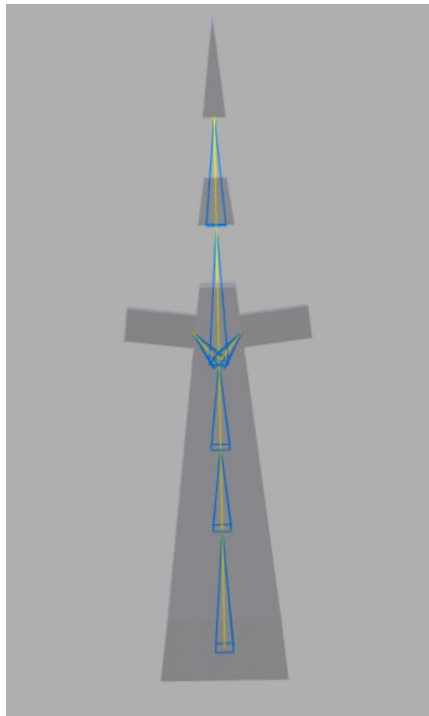


Figure 9 Rig from Figure10 displayed in Open3mod, assimps model viewer

Because of the way skeletons are defined there's no usual way for us to display the leaf bone as displayed in the modeling application (Figure 14), this is why to display the leaf bone the

application exporting the skeleton should also have a way of adding a joint in the place of the tail of the leaf bones.

In the case of blender, the fbx exporter has an “add leaf bones” option, which adds another joint that has no animation and is not controlling any part of the mesh at the ends of the skeleton (Figure 15).

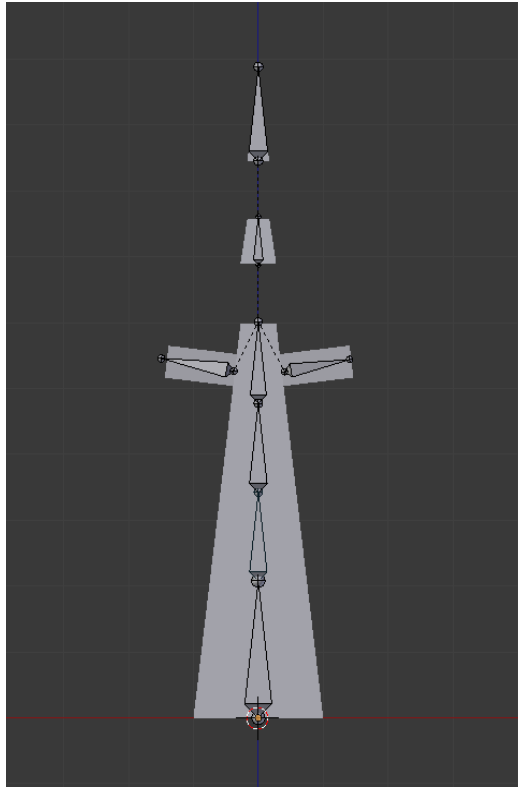


Figure 10 Before adding leaf joints

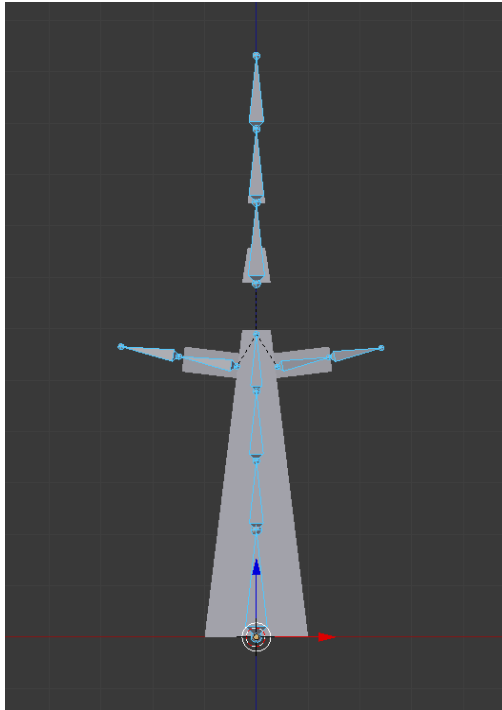


Figure 11 After adding leaf joints

In our application, we then can use these joints to display the leaf bones (Figure 16). Since they are not controlling any mesh we recognize them and add them by name (they are suffixed with “_end”).

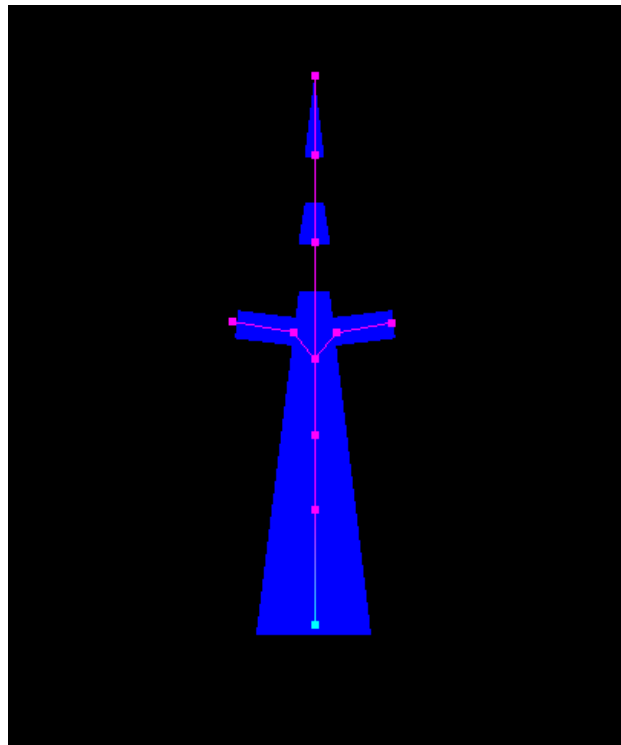


Figure 12 Final visualization in our application, this mesh has added leaf bones.

4.8. Animating and interpolation

Once the application starts the OpenGL loop and displays loaded assets on the screen, the user can choose to playback an animation, and control the playback of that animation.

The OpenGL render loop draws a scene into a buffer and then displays that buffer on the users' screen, we continually repeat this process until exiting the application. This one drawn image is called a frame, the length of time in between drawing of frames is called delta time, and the number of frames drawn in one second is called framerate.

To properly playback the animations we need to make it independent of our applications framerate, in other words it shouldn't be faster or slower if our applications framerate is faster or slower. Animations have their own internal framerate which we refer to as animation framerate; an animation can be created as a 25 frame per second animation but played back in any chosen framerate. Playing back the animation at higher framerates is possible by interpolating in between frames of the original animation.

To determine the transformation of bones when drawing a frame, we must find the time since the start of the application. The running time of the animation is the current time minus the time at which the animation started. Other controls like pausing or moving the animation by frame are achieved by manipulating the start time of the animation; in case of pausing we move the start and current time further in time by adding the delta time, while iterating frames, we subtract or add the time of one frame to the start and current times.

```
fsec RunningTime = (tAnimationCurrent - tAnimationStart);
timeInSeconds = RunningTime.count();

float ticksPerSecond = scene->mAnimations[animation]->mTicksPerSecond != 0 ?
    scene->mAnimations[animation]->mTicksPerSecond : 25.0f;

float timeInTicks = timeInSeconds * ticksPerSecond;
//loops the animation
float animationTime = fmod(timeInTicks, scene->mAnimations[animation]-
    >mDuration);
```

ticksPerSecond is the animations predefined framerate, timeInTicks is how much ticks have passed since the start of the animation, finally animationTime is timeInTicks but modded by the duration, modding the time by its duration makes the animation loop.

Once the time of the animation is determined we have to calculate the transformation of all the bones in the hierarchy. This is done in the recursive function ReadNodeHierarchy


```

void Model::ReadNodeHeirarchy(float animationTime, const aiNode * pNode,
mat4x4 parentTransform, int animation)
{
    std::string nodeName(pNode->mName.data);

    int animationBuffer = animation;
    float animationTimeBuffer = animationTime;

    aiAnimation* pAnimation = scene->mAnimations[animation];

    mat4x4 nodeTransformation;
    mat4x4_identity(nodeTransformation);
    aiMatToLinMat(nodeTransformation, pNode->mTransformation);

    const aiNodeAnim* pNodeAnim = nullptr;

    pNodeAnim = animations[pAnimation->mName.data][nodeName];

    if (BoneMasks.find(nodeName) != BoneMasks.end()) { // mask starts here
        pAnimation = scene->mAnimations[BoneMasks[nodeName].animation];
        //switch out the animation
        animation = BoneMasks[nodeName].animation;
        animationTime = BoneMasks[nodeName].animationTime;
    }

    //if this node is an animated bone
    if (pNodeAnim && animate == true) {

        //interpolate scaling and generate scaling transformaion matrix
        aiVector3D scaling;
        CalcInterpolatedScaling(scaling, animationTime, pNodeAnim);

        mat4x4 scalingM;
        mat4x4_identity(scalingM);
        mat4x4_scale_aniso(scalingM, scalingM, scaling[0], scaling[1],
scaling[2]);

        //interpolate rotation and generate scaling transformomration matrix

        aiQuaternion rotationQ;
        CalcInterpolatedRotation(rotationQ, animationTime, pNodeAnim);
        aiMatrix3x3 rotationAiMat = rotationQ.GetMatrix();
        mat4x4 rotationM;
        mat4x4_identity(rotationM);
        quat rotQuat;
        rotQuat[0] = rotationQ.x;
        rotQuat[1] = rotationQ.y;
        rotQuat[2] = rotationQ.z;
        rotQuat[3] = rotationQ.w;
        //quat_norm(rotQuat, rotQuat);
        mat4x4_from_quat(rotationM, rotQuat);

        //interpolate translation and generate translation transformation
matrix
        aiVector3D translation;
        CalcIntrepolatedPosition(translation, animationTime, pNodeAnim);
    }
}

```

For every node in the hierarchy that is not a bone we determine its transformation by multiplying its parents' transformation matrix and its transformation matrix relative to the node parent, and pass that matrix to its children.

If the node is a Bone the procedure differs.

We need to determine the transformation of the bone defined by the animation.

We calculate the scale, rotation and position of the bone and combine that into a transformation matrix.

The bones total transformation is its transformation matrix defined by the animation that's in its bone space, combined with all the transformation matrices of its parents, as bones are hierarchical and they influence their children with their own transformation.

We left multiply the transformation with the bone offset matrix, which converts vertices to bone space to get the final transformation that we sent to the graphic pipeline.

4.8.1 Interpolation

First we need to obtain the transformations of the animation by interpolating the position, rotation and scale keyframes.

If our animation time and our animation do not fall on any of the edge cases, like the time preceding or exceeding animation time, or having too few keyframes we will need to determine the transformation by interpolating keyframes.

```
GLuint Model::FindPosition(float animationTime, const aiNodeAnim * pNodeAnim)
{
    assert(pNodeAnim->mNumPositionKeys > 0);

    for (GLuint i = 0; i < pNodeAnim->mNumPositionKeys; i++) {
        if (animationTime < (float)pNodeAnim->mPositionKeys[i + 1].mTime)
        {
            return i;
        }
    }
    assert(0);
}
```

To interpolate between keyframes first we need to find in between which two keyframes our current time lies.

We iterate over the keyframes via their indexes, and once the i-th + 1 keyframe in the iteration surpasses the animation time, we know our animation time lies in between keyframe i and i+1.

```

void Model::CalcIntrepolatedPosition(aiVector3D & out, float animationTime,
const aiNodeAnim * pNodeAnim)
{
    if (pNodeAnim->mNumPositionKeys == 0) {
        out.x = 0.0f;
        out.y = 0.0f;
        out.z = 0.0f;
        return;
    }

    if (pNodeAnim->mNumPositionKeys == 1) {
        out = pNodeAnim->mPositionKeys[0].mValue;
        return;
    }

    if (animationTime <= pNodeAnim->mPositionKeys[0].mTime) {
        out = pNodeAnim->mPositionKeys[0].mValue;
        return;
    }

    if (animationTime >= pNodeAnim->mPositionKeys[pNodeAnim-
>mNumPositionKeys - 1].mTime) {
        out = pNodeAnim->mPositionKeys[pNodeAnim->mNumPositionKeys -
1].mValue;
        return;
    }

    GLuint positionIndex = FindPosition(animationTime, pNodeAnim);
    GLuint nextPositionIndex = (positionIndex + 1);
    assert(nextPositionIndex < pNodeAnim->mNumPositionKeys);

    float factor = (animationTime - (float)pNodeAnim-
>mPositionKeys[positionIndex].mTime) /
        ((float)pNodeAnim->mPositionKeys[nextPositionIndex].mTime -
(float)pNodeAnim->mPositionKeys[positionIndex].mTime);

    assert(factor >= 0.0f && factor <= 1.0f);

    const aiVector3D& startPosition = pNodeAnim-
>mPositionKeys[positionIndex].mValue;

    const aiVector3D& endPosition = pNodeAnim-
>mPositionKeys[nextPositionIndex].mValue;

    //linearInterpolaiton LERP
    out = startPosition * (1.0f - factor) + endPosition * factor;
}

```

To find calculate the interpolation we use linear interpolation for scale and position and spherical interpolation for rotation.

The formula is such:

$$factor = \frac{animationTime - keyframe[i].time}{keyframe[i + 1].time - keyframe[i].time}$$

$$position = keyframe[i].position * (1 - factor) + keyframe[i + 1].position * factor$$

Once we calculate the transformation matrix, we obtain the global transformation by multiplying with the parent transform, this is passed down to the children.

$$globalTransformation = parentTransformation * transformation$$

The final transformation of the bone is obtained

$$globalTransformation * boneOffset$$

This transformation will be passed to the shader to determine the final position of vertices.

4.9. Adding masks

Masking animations is the process of playing another animation on a designated part of the skeleton. For example, a character can be playing a walking animation, but we can mask the head and play the look around animation, or make the right hand hold an item.

In the application, the user specifies a bone which will be the root of the mask, all children of that bone are also masked, for example picking the left shoulder will mask the entire left arm.

For the masked bones, we simply use the masked animation instead of the original. The masked animation has its own playback time that should be independent from the original animation, for example a walk cycle can be shorter than the masked look around animation, so each animation should store its own start and current times.

4.10. Blending animations

Blending animations is the process of combining two animations in a way that results in an animation that is a mix of the two.

This technique is used to create smooth transitions between animations such as transitioning from standing to walking or to create new animations that are a mix of two.

In our application, we can combine two imported animations into a transition or a weighted blend.

4.10.1. Animation Blend

A weighted blend is an animation that is a combination of two animation with a weight in range 0 to 1. 0 meaning the first animation is playing and 1 meaning the second is playing, any value in between will mix the two.

```
//interpolate scaling and generate scaling transformaion matrix
aiVector3D scaling1;
CalcInterpolatedScaling(scaling1, animationTime, pNodeAnim);

aiVector3D scaling2;
CalcInterpolatedScaling(scaling2, animationTime2, pNodeAnim2);

aiVector3D scaling;
//lin interpoalte scaling
scaling = scaling1 * (1.0f - factor) + scaling2 * factor;

mat4x4 scalingM;
mat4x4_identity(scalingM);
mat4x4_scale_aniso(scalingM, scalingM, scaling[0], scaling[1], scaling[2]);

//interpolate rotation and generate rotation transfomration matrix
aiQuaternion rotationQ1;
CalcInterpolatedRotation(rotationQ1, animationTime, pNodeAnim);

aiQuaternion rotationQ2;
CalcInterpolatedRotation(rotationQ2, animationTime2, pNodeAnim2);

aiQuaternion rotationQ;
aiQuaternion::Interpolate(rotationQ, rotationQ1, rotationQ2, factor);
rotationQ = rotationQ.Normalize();

aiMatrix3x3 rotationAiMat = rotationQ.GetMatrix();
mat4x4 rotationM;
mat4x4_identity(rotationM);
quat rotQuat;
rotQuat[0] = rotationQ.x;
rotQuat[1] = rotationQ.y;
rotQuat[2] = rotationQ.z;
rotQuat[3] = rotationQ.w;
//quat_norm(rotQuat, rotQuat);
mat4x4_from_quat(rotationM, rotQuat);

//interpolate translation and generate translation transformation matrix
aiVector3D translation1;
CalcIntrepolatedPosition(translation1, animationTime, pNodeAnim);

aiVector3D translation2;
CalcIntrepolatedPosition(translation2, animationTime2, pNodeAnim2);

aiVector3D translation;
translation = translation1 * (1.0f - factor) + translation2 * factor;

mat4x4 translationM;
mat4x4_identity(translationM);
mat4x4_translate(translationM, translation.x, translation.y, translation.z);

mat4x4_scale_aniso(translationM, translationM, scaling[0], scaling[1],
scaling[2]);
mat4x4_mul(nodeTransformation, translationM, rotationM);
```

We achieve this by finding the interpolated transformation of both animations as we normally would. Then we interpolate their calculated position, scale and rotation using the user specified weight before combining them into a transformation matrix.

The result should be a mix of the two animations, both animations are looped by their respective duration.

4.10.2. Transitions

Transitions are achieved much in a similar way as blends.

The key difference is that instead of weight we specify the starting frame of the transition and the ending frame of the transition. Since animation durations are measured in frames it is easier to determine where we want the transition to begin and end by looking at the frames of an animation.

The transition start and end frames do not need to be necessarily positioned inside the duration of the first animation, they can begin and end outside, though a general practice is to position the beginning during the first animation and the ending during, or after the first animation.

```
fsec RunningTime = (transitions[animateSpecial].tAnimationCurrent -
transitions[animateSpecial].tAnimationStart);
timeInSeconds = RunningTime.count();

float ticksPerSecond1 = scene->mAnimations[anim1]->mTicksPerSecond != 0 ?
scene->mAnimations[anim1]->mTicksPerSecond : 25.0f;

float ticksPerSecond2 = scene->mAnimations[anim2]->mTicksPerSecond != 0 ?
scene->mAnimations[anim2]->mTicksPerSecond : 25.0f;

float transitionInTicks = frameEnd - frameStart;

//animations should have the same ticks per second
float ticksPerSecond = ticksPerSecond1;

//total ticks per second aka duration is anim1 + anim2 - transition;
float duraiton = scene->mAnimations[anim1]->mDuration + scene-
>mAnimations[anim2]->mDuration - (transitionInTicks);

float timeInTicks = timeInSeconds * ticksPerSecond;
//loops the animation
float animationTime = fmod(timeInTicks, duraiton);
```

We calculate the total duration as duration of the first animation + duration of the second animation – transition duration.

During playback of the transition we play three parts, the first animation, the transition itself, and the second animation.

The result will be an animation that is gradually transitioning to the next at specified times.

The first part is straightforward; we simply play the original animation up until the start frame of the transition.

During the transition, we blend the animations, our weight is interpolated based on the duration of the transition.

$$weight = \frac{animationTime - duration}{transitionInTicks}$$

At the start of the transition the weight is 0, then it gradually increases until we reach the end of the transition where it becomes 1.

The last part is also straightforward, we simply play the second animation, its start time is the start time of the transition.

5. Application Use

5.1. Foreword on importing the model

The developed application uses the Assimp import library to import various file types, and therefore should share the same format support as the Assimp importer.

However this application has only been tested on .obj, .fbx and .dae (collada) files.

Careful considerations should be taken when preparing a model to be imported.

This application does not display materials with solid colors, it only displays textures, the imported model should be fully textured.

If the model does not have any textures or fails to load them, it will appear black, however it is still possible to see it by changing the render mode to Weights or AllWeights.

When exporting the model to a format supporting a skeleton system if it is not triangulated it is must be done in the application (.fbxs are likely to need this). Do this by calling the function `LoadAssimp(path_to_file, true);` in main.

You should check the “apply modifies option”.

You should check the “add leaf bones” option if available, said leaf bones should be suffixed with a “_end” by the exporter, otherwise do this manually or they will not display. However, the model should animate correctly without them.

The model prior to exporting should be in its bind pose, as assimp takes the first frame currently playing as the bind pose of the skeleton.

If the model is oriented wrongly consider changing the forward and up axes in the exporter of your choice as many programs define forward and up axes differently from OpenGL.

If you find that the application lags and the animations reproduced are slower, consider whether you are running the application on a discrete GPU and whether its specifications are viable.

5.2. Application Features

Once the application loads there are several possible options to explore.

There are 3 different render modes, an unshaded textured mode (Figure 7), a mode that displays the total of all weights on the mesh (Figure 8), and a mode that displays the weight

of a single bone and allows iteration over bones (Figure 9). The skeleton can be displayed in any mode (Figure 10).



Figure 13 Unshaded render mode

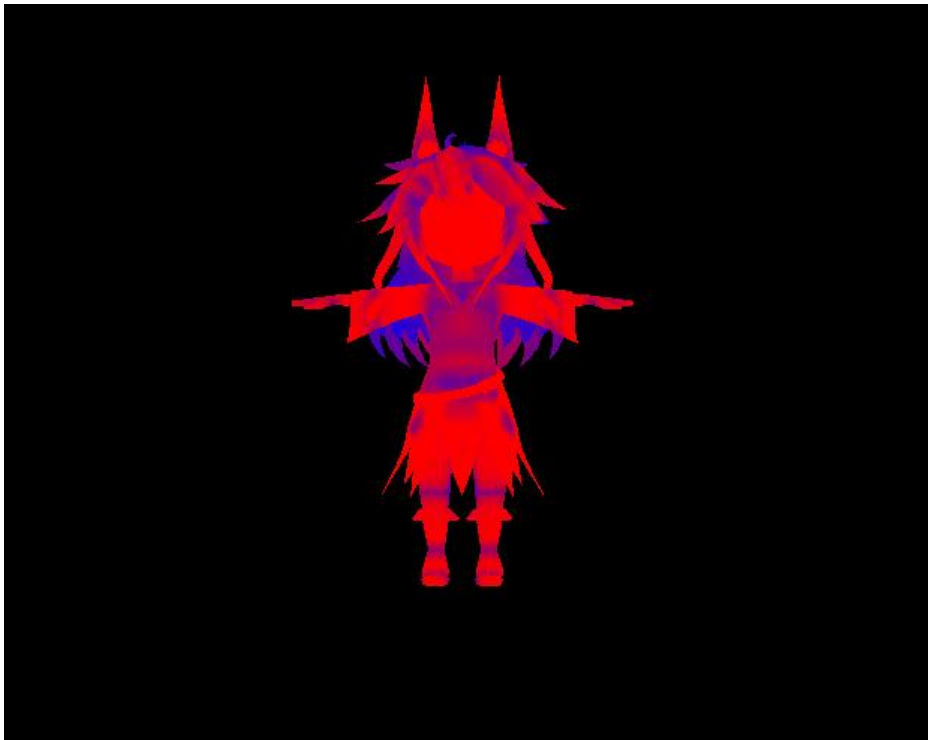


Figure 14 Total Weights of the model, displaying the highest value weight on the mesh, red being 1 and blue being 0 (no influence), the bluish parts of the mesh appear near joints where the values are mostly 0.5 due to being controlled by 2 bones.

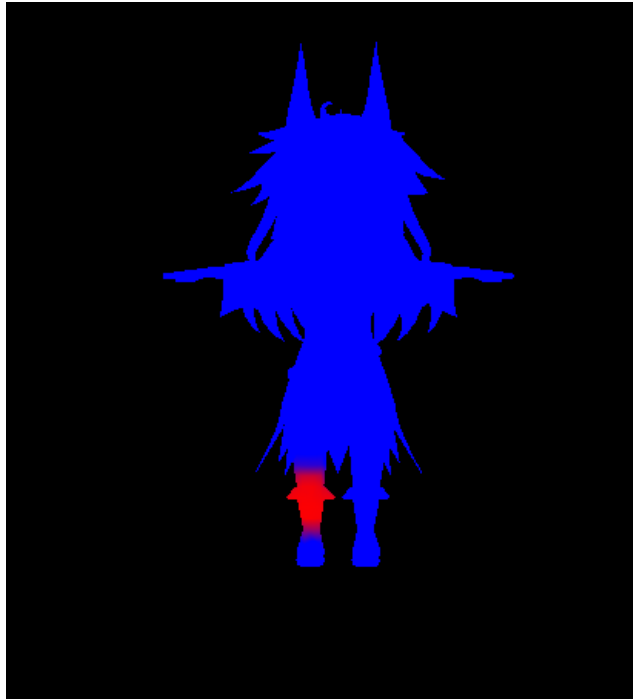


Figure 15 Weights of a single bone

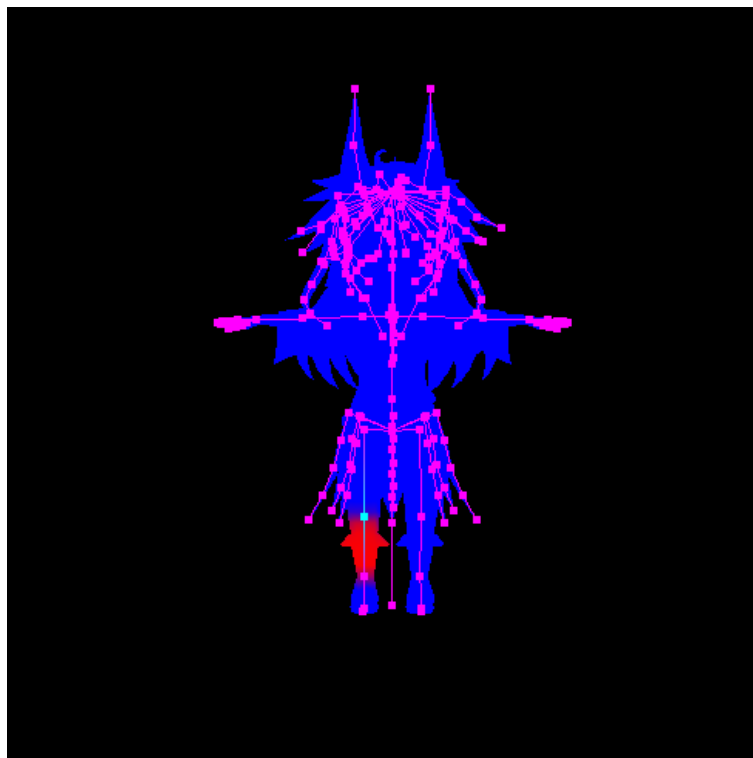


Figure 16 Weights of a single bone with the skeleton displayed, the blue dot is the selected bone, or more accurately joint, transformations on this joint influence the red area, and the child bones.

Animations that are imported through assimp can be played, paused, iterated frame by frame or by 10 frames backwards and forwards.

The application can be controlled through the console where it allows several features such as making animation Blends, Transitions and Masks, and other controls.

5.3. Application manual

The following controls can also be found in the main.cpp file

Camera controls:

W A S D – move
SHIFT + mouse -> look around
CTRL + mouse -> pan
ALT + mouse -> orbit around look at point
scroll -> zoom in/out

Animation Controls:

0 -> turn off animations
1 -> turn on animations / play next animation
P -> play / pause animation
right arrow -> next frame (pauses the animation and advances a frame)
left arrow -> previous frame
CTRL + right arrow -> skip 10 frames
CTRL + left arrow -> go back 10 frames
B -> toggle playing blends or regular animations
Up arrow – increase the weight when playing an animation blend
Down arrow – decrease the weight when playing an animation blend

Render Controls:

Press K to toggle render modes -- UNSHADED, ALLWEIGHTS and WEIGHTS per bone
SHIFT + . -- to iterate over bones only in render type WEIGHTS
Press L to toggle armature

Console Commands:

F1 – start console
"playAnimation" -- invoke an animation by name
"selectBone" -- select a bone in bone weight render mode to display its weights
"turnOnMask" – create a mask by choosing a Bone from which the animation plays, and then the animation that will Play
"clearMasks" – removes the current mask
"createAnimationBlend" – create and store an animation blend (indices start at 0)
"createTransition" – create and store a transition (indices start at 0)
"playAnimationBlend" – play an animation blend by its index
"listAnimations" – lists imported animations by name (animations with the prefix "rig|" should be typed without it)

6. Conclusion

Skeletal animation is a powerful 3D animation tool that is essential for creating 3D animation today. It is a primary tool for animation in the movie and game industry and many others and is often used in conjunction with other animation techniques.

Skeletal animation is not supported by many file extensions, the most widespread extension .fbx is owned by Autodesk, made for Autodesk software and it is closed source, however it is also used by most other software that uses 3D objects, materials and skeletons, such as game engines, modeling software, 3D generative software.

Another widespread file type is the collada (.dae) type, meant to be a true exchange format and open source, however it does not provide an SDK so applications are required to implement the specifications themselves.

For both .fbx and .dae are extremely complex formats and no third party application has implemented the full specifications, resulting in a lot of bugs when exporting and importing files using these formats across applications.

The program used to export the model and skeleton Blender has its own fbx and collada exporter/importer, however both are buggy and not quite complete, due to this fact when models exported from third party apps can have several unforeseen bugs.

Due to the high complexity associated with file formats supporting skeletons, and the many problems with writing an importer, writing a file extension from scratch or writing an importer for an existing one wasn't in the scope of this project, so we have chosen to use Assimp (Open Asset Import Library) to import various formats successfully.

Our project was to create an application that can support skeletal rigs and their animations, reconstruct the skeleton in the application, create functions that use interpolation techniques to accurately reproduce animations in real time, explore techniques of masking and blending animations.

7. BIBLIOGRAPHY

[1] "Skeletal Animaiton", Wikipedia: The free encyclopedia. Wikimedia Foundation

https://en.wikipedia.org/wiki/Skeletal_animation [05.06.2017]

[2] Etay Meiri. Skeletal Animation With Assimp,

<http://ogldev.atSPACE.co.uk/www/tutorial38/tutorial38.html> [05.03.2017.]

[3] Joey de Vries, OpenGL

<https://learnopengl.com/#!Getting-started/OpenGL> [05.02.2017]

[4] Joey de Vries, Hello Triangle

<https://learnopengl.com/#!Getting-started/Hello-Triangle> [05.02.2017]

[5] Alexander Overvoorde

<https://open.gl/drawing> [06.06.2017]

[6] Joey de Vries, Coordinate Systems

<https://learnopengl.com/#!Getting-started/Coordinate-Systems> [05.02.2017]

[7] Lars Pensjö, Doing animations in OpenGL, 21 June 2012

<http://ephenationopengl.blogspot.hr/2012/06/doing-animations-in-opengl.html>

[06.06.2017]

[8] Alexander Gessler, Thomas Schulze, Kim Kulling, David Nadlinger, assimp - Open Asset Import Library July 2012

http://assimp.org/lib_html/data.html [01.02.2017]

Human Skeleton Animation System

Abstract

In this paper, we explain the basics of skeletal animation, the representation and meaning of data it requires and the procedures for its reproduction and manipulation. We explain the structures and specifications of the *Assimp* library and the techniques for extracting necessary data and reorganizing it for further use. It details the implementation of algorithms for reproducing animation via interpolation, masking and blending animation, constructing a visual representation of the skeleton and other related algorithms.

The purpose of this project is to explore and explain the behind the scenes algorithms of skeletal animation, and provide an OpenGL implementation of a skeleton system and support animation playback.

Keywords: OpenGL, skeleton, rig, armature, Assimp, animation, interpolation, blending, masking, C++.

Animacija skeletnog modela čovjeka

Sažetak

U ovom radu objašnjene su osnove skeletne animacije, reprezentacija i značenje potrebnih podataka i procedure za njegovu reprodukciju i manipulaciju. Objašnjene su strukture i specifikacija *Assimp* biblioteke i tehnike za ekstrahiranje potrebnih podataka i reorganizacija tih podataka za daljnje korištenje. Opisane su implementacije algoritma za reprodukciju animacije interpolacijom, maskiranje i miješanje animacija, konstrukcija vizualne reprezentacije kostura i ostali potrebni algoritmi.

Svrha ovog projekta je istraživanje i objašnjenje prikrivenih algoritama skeletne animacije i pružanje OpenGL implementacije skeletnog sustava i podrška za reprodukciju animacija.

Ključne riječi: OpenGL, kostur, Assimp, animacija, interpolacija, miješanje, maskiranje, C++.