DePaul University 2019 Autumn - 2020 Spring

# GAME ENGINE PROGRAMMING AND REAL TIME SYSTEM

Game Engine
&
FBX Converter

Xuefan Wang

# Part 1: Game Engine

## 1. Library (static/ dynamic)

### 1.1 File System:

Using **16 bytes** data alignment: Good for 4x1 vector or 4x4 matrix.
Using my own heap and memory header to define malloc and free function.
Use my own file header and **secret pointer to accelerate** malloc and free speed.
**Result: 4x faster** than original.

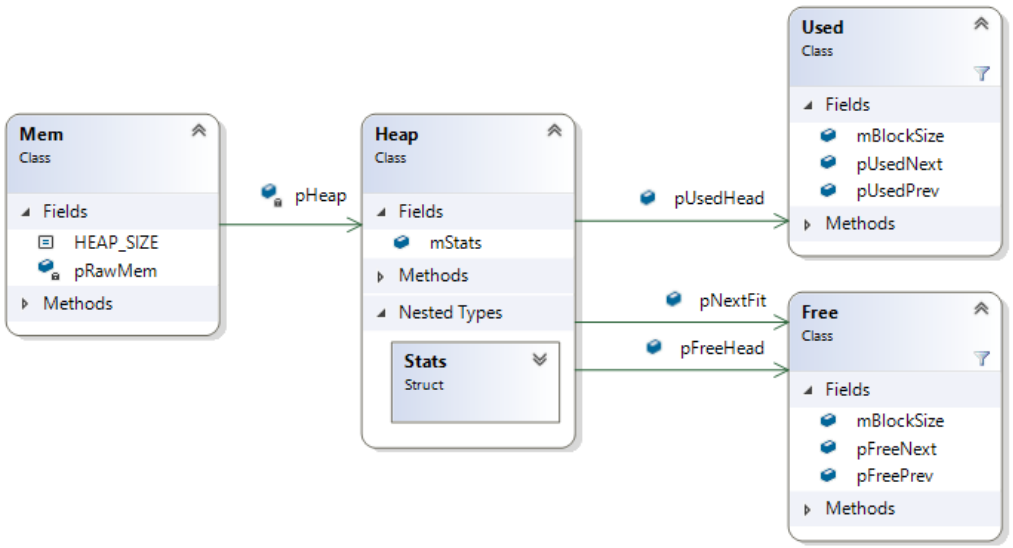


Figure 1: UML of File System

### 1.2 Math lib:

Implementing all functions of vector, matrix, and quaternion.
Defining all methods needed for the mathematics related to them.
Implementing **Lerp and Slerp** function for game engine.
Using **SIMD** to do parallel computing on CPU.
**Result:** Saving **25%-40%** of time.

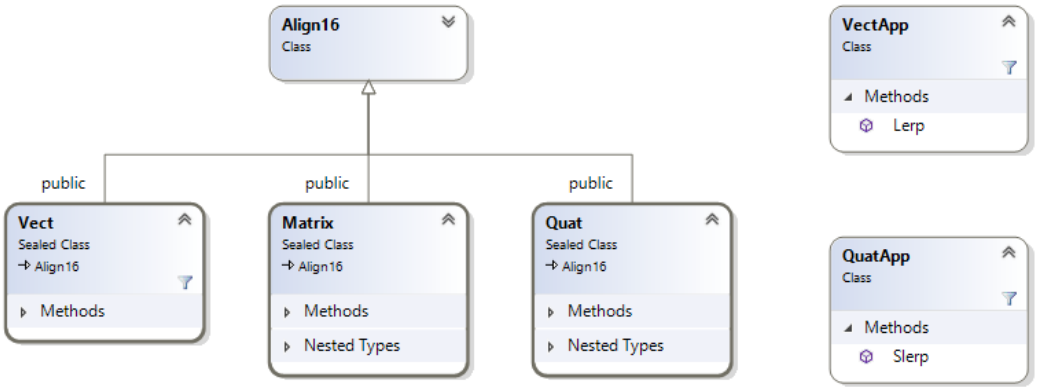


Figure 2: UML of Math Library

### 1.3 OpenGL and sb7:

GLFW is imported from open source code.

Sb7 is imported from OpenGL super bible 7th edition.

### 1.4 Time:

Time and Timer is imported from open source code.

### 1.5 Data structure:

Double Linked List (**DLink**) is the basic for all other structures.

N-ary Tree based on DLink.

Using iterating structure to find nodes in the tree. DLink helps to find prev or next siblings and parents in O(1).

**Result:** Both inserting and deleting time is **O(1)**. Saving iteration time especially **backward iteration** time to **O(n)**.

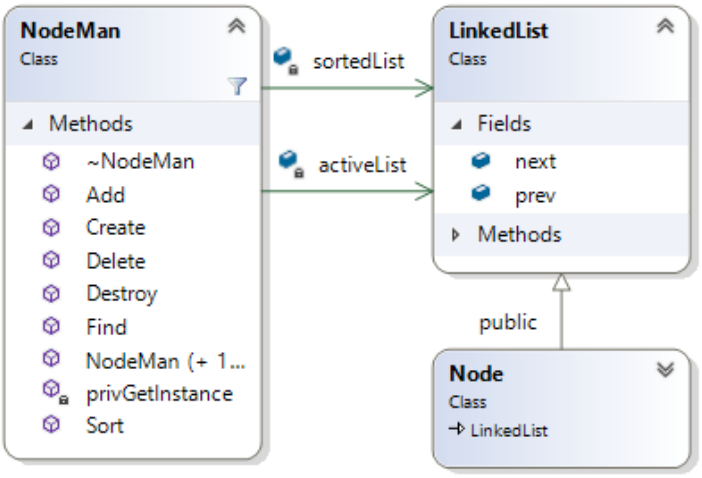


Figure 3: UML of DLink structure

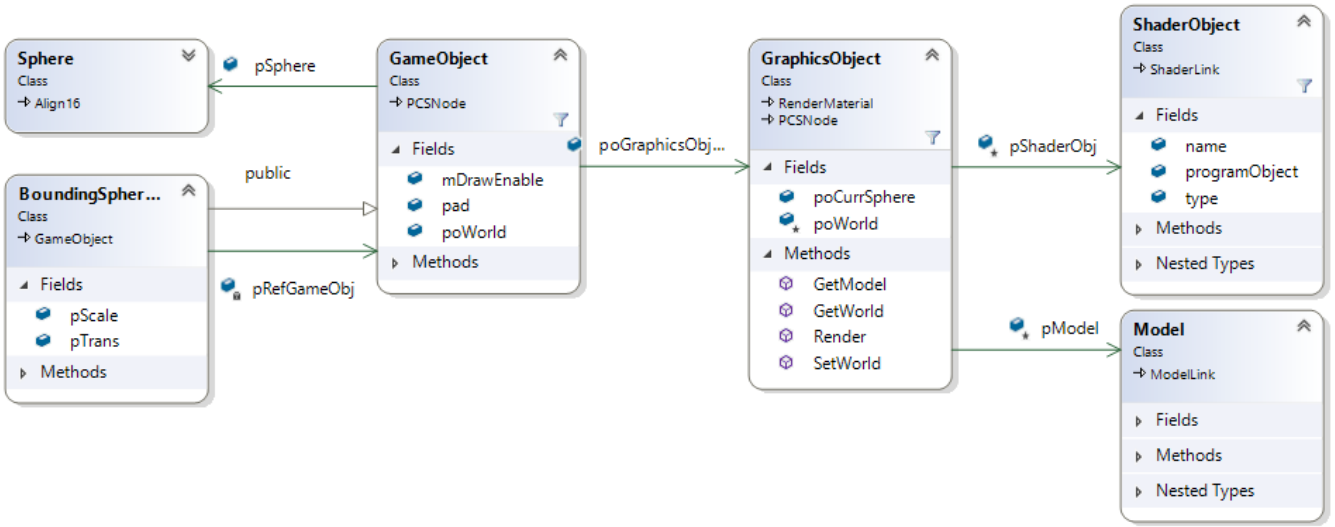


Figure 4: UML of Game Objects

## 2. Game Systems:

### 2.1 Game Objects:

Model is to keep all the triangles of vertex and color information.
Shader is to tell how GPU renders a model.
Graphic Object is to combine model and shader.
Game Object is to wrap Graphic objects with user interface.
**Result:** User of the engine do not have to know the details of OpenGL function.

### 2.2 Camera:

3D perspective Camera for 3D model (Most game objects)
2D orthographic camera for 2D model and fonts

### 2.3 Bounding sphere:

To surround each game object with a bounding sphere when loading game, this is for **Collision testing**, which saves time when running in real-time system.
Using a hierarchy structure of bounding sphere to detect collision.

### 2.4 Input System:

Game controlling system with keyboard and/or mouse.

### 2.5 Font System

Every fontObj is a special game object, and each message is a list of font objects.
Each glyph is a subclass of image class, and is stored in glyph manager using object pools.

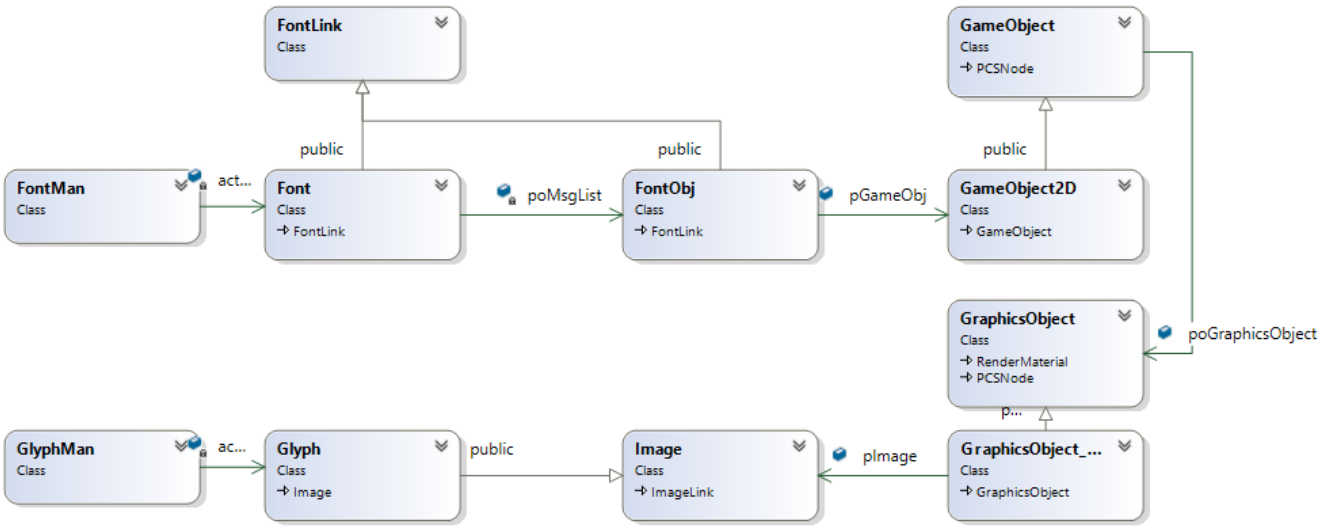


Figure 4: UML of Fonts

## 3. Animation System:

### 3.1 Skeleton:

Rendering data from hierarchy of bones based on their relative movement.
Using keyframes from data extracted from .fbx file.

Using lerp and slerp to calculate real time bone position.

Local position = Translation * Rotation * Scale

For this part, I use load in place technique, to save loading time. However, file size is extremely large because all data needed to run the program is in the file.

World position = parent world position * local position

### 3.2 Skin:

Skin vertices is affected by its related bones.

$$u_{(t)} = \sum_{i=0}^{n-1} \omega_i B_{i(t)} M_i^{-1} p \text{ where } \sum_{i=0}^{n-1} \omega_i = 1 \text{ (function 1)}$$

Since most of the skins are affected by less than 4 bones, and for those affected by more than 4 bones, the extra data has little effect on the bone. So that, in my engine, I choose n = 4.

Form .fbx file, we extract the (1)bone weight table, (3) invPose matrix, and (4)all the vertices in the skin model. The (2) bone position matrix is calculated in 3.1.

Combining all the data together we get the skin matrix at time t.

### 3.3 Computer Shader:

Since most of calculations from animation is float point multiplication, moving all calculations to GPU is far more efficient than calculating with CPU.

Instead of using vertex shader before transferring data to fragment shader, we use computer shader to calculate all the vertices based on the function 1.

#### 3.3.1 World Compute Shader

Loading all data in part 3.1 to SSBO (Shader Storage Buffer Object), and also, we need to set up a magic table to let the GPU know what parent matrices are needed to get local position.

#### 3.3.2 Mixer Compute Shader

Using the data from world shader, we can now load all data of function 1 to another SSBO, to calculate the skin vertices. After that, we pass the vertices to a fragment shader for rendering.

## 4. Reducing redundant data

In 3.1 we use load in place technique; however, the negative part is large file size.

Since most models do not really use scaling, and the translation are always the same except hip bone, we can remove the redundant data from frames and reduce the animation file size to **33.3%** of its original size.

Even more shrinking the file size is in FBX Converter Part 2: 1.4

# Part 2: Film box (FBX) Converter:

To render a .fbx file in my own engine, we have to get all the necessary data from it, and put those data in a binary file. The layout of the file and data needed will be discussed in this section.

## 1. Model and Skeleton Data

### 1.1 Model vertices and triangles

All data of control points and triangles are stored in Mesh.

We can get all control points based on the triangle. However, since each control points might be shared by several triangles, we can eliminate duplicated data by sort. This step reduces the file size by at least (**50%**).

Since all vertices are sorted, the indices are also changed, so triangle indices also need to be modified.

### 1.2 Bone Hierarchy

Each bone position is relative to another bone, so knowing all the hierarchy is necessary. The data is stored in Hierarchy. This part is simple, but we need to know the depth of each bone for the magic table mentioned in 3.3.1.

### 1.3 Key Frame

All the bone poses of each key frame are stored in Animation as a curve.

Each .fbx file may store several stacks of animation curve, and each stack may have different size of key frames, each key frame has same size of bone poses.

### 1.4 **Reduce Key Frame**

Since same key frame can be calculated by its previous and afterward keyframes, we can drop those key frames safely. And other key frames are very similar with the calculated one, which is undetectable by human eyes, we can also drop those to save space.

In my FBXConverter, I arbitrary choose less than 5% differences, which reduces the file size by **10~20%.**

## 2. Skin Data

$$u_{(t)} = \sum_{i=0}^{n-1} \omega_i B_{i(t)} M_i^{-1} p \text{ where } \sum_{i=0}^{n-1} \omega_i = 1 \text{ (function 1)}$$

### 2.1 Weight ($\omega_i$)

Each bone cluster has all the values for skin rendering. and they are stored in Link. It has all the skins with weights corresponding to the bone. However, trying to use all the data will give us a large binary file and also takes more time to calculate in real time.

So that, we can choose the **first 4 bone weights** that affect skin, which give us good simulation and also reduce the file size by **5~10%**.

### 2.2 Bone position ($B_{i(t)}$)

Bone position at time t is discussed in Part 1: 3.1 skeleton and Part 2: 1.2 Key frame.

### 2.3 Inverse pose ($M_i^{-1}$)

Transform and link matrix are also stored in bone cluster, we can calculate $M^{-1}$ by using transform matrix * inverse link matrix.

And now, instead of saving 2 matrices, we only save 1 matrix to the binary file, which could

reduce the file size by **50%.**

### 2.4 Skin vertex ($p$)

Original Skin vertex is the one we have discussed in Part 2: 1.1

With all the data available now, we can render skin in real time.

## 3. File Layout

### 3.1 Header File

Different data have different composition and layout. Using header file to indicate what data layout is, and what is the corresponding offset.

Since I put all data into one binary file, so we have different sections of data. At the beginning of the file, a header of headers is also added, so that we can easily file the file location without **no delay**.

### 3.2 Data

Only raw value is stored in a binary file, no data structure is in there, the only way to read the data is using the header file and knowing the offset and layout.

### 3.3 Data Layout Example(Skeleton):

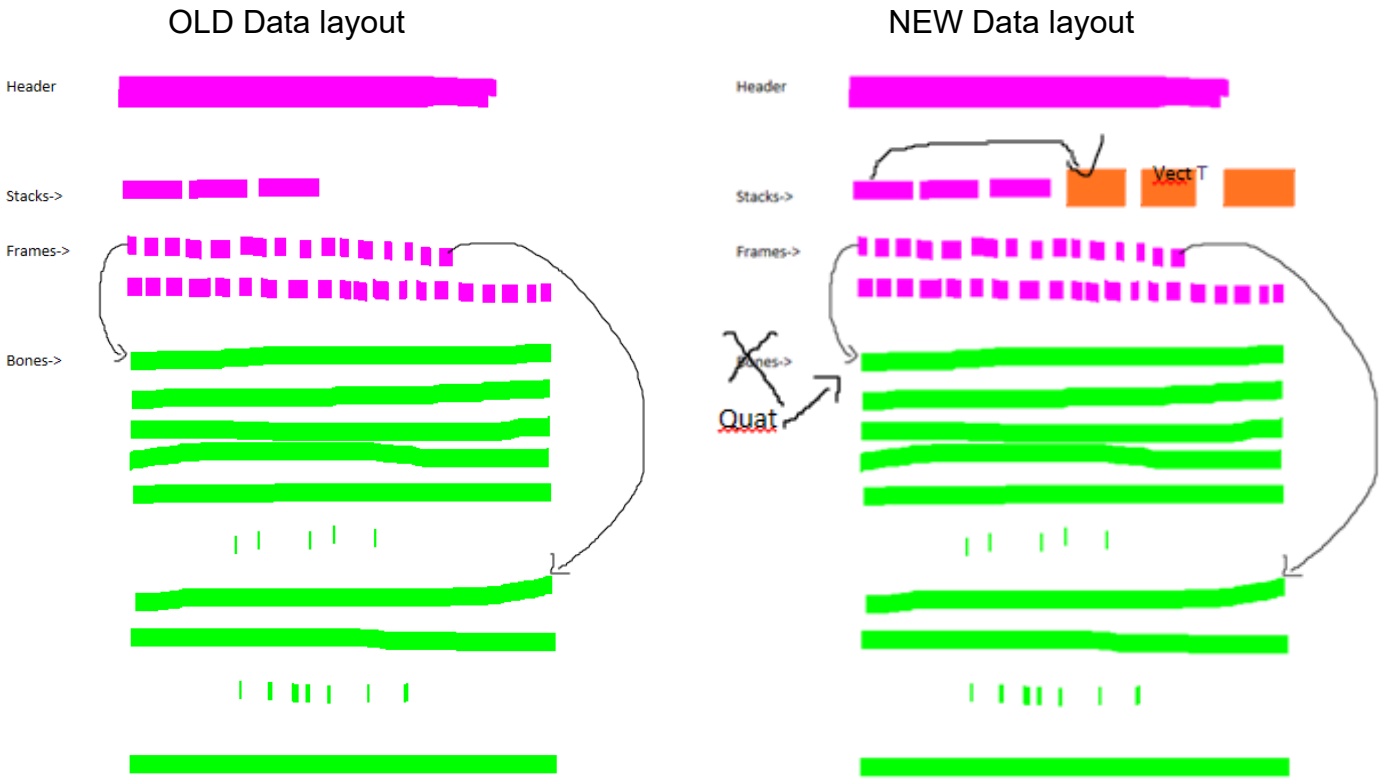


Figure 5: Data Layout

We have 4 parts in this file: header, stacks, frames, and bones.

In the header file, we have the whole file size, and 3 offsets showing where each section starts.

Each stack saves how many frames it has, so that, each stack could easily find where its staring frame is.

In the old data layout, each Frame has a pointer to its bone list, and each bone has 3 4x4 matrices. Total size is 3 * 16 * sizeof(float) = 192 bytes

In the new data layout, each Frame has a pointer to only rotation matrix, which is 16*sizeof(float) = 64 bytes. This step reduces the file size by 66%.

Also, In the new data layout, we can drop some keyframes, by calculating from nearby frames, this step can drop 1/10 of key frames, so reduce the file size by 10%.

All the data layout is defined in its own struct, so the Converter and Game engine must have the same struct. If one of them is updated, the other one should be updated also. Otherwise, garbage data will be loaded into the engine.

## Part 3: Conclusion

This is the basic setup and usage of my game engine, it has 2D and 3D rendering function. With reading data from .fbx file, we can easily run it in our own engine with reduced file size and faster speed.

Example of using this engine to develop a real time game is my Space Invaders program, which is written in C#.