**CS560 – Animation: Hierarchical Model and**

**Key Frames Animation**

1. **Instructions to Build and Run the application**
2. Unzip the submission file and save it in your preferred folder.
3. It is strongly recommended to use a path with small length; issues may appear with paths longer than 200 characters.
4. Under the graphics folder, double click the Graphics.sln solution file.
5. Go to Build->Rebuild Solution. It is recommended to build the application in release mode, because in debug mode in slower machines framerate could be affected. However, the test results in DigiPen’s PCs showed the application running at 60FPS in average, in both debug and release modes.
6. Once the solution is built, under the graphics folder, execute **copy\_files\_debug.bat** and/or **copy\_files\_release.bat**.
7. Now execute the application whether within visual studio or using the .exe files generated in the **Graphics\Debug** or **Graphics\Release** folders.
8. **Modeling**

***Complexity of the object***

The object used in my implementation was the sample model named **“tad.fbx”** available in the course website, which actually contains 65 bones.

***Hierarchical structure representation and implementation:***

*Outline of the engine:* It was used a Game Engine that uses a Graphics API, both created in my own. Since the Graphics API was designed to be used by any game engine architecture paradigm, it has one important class named **GLObject**, which is the “connection” to the model information, it means that, in order to create and draw a model in a scene, the application needs to call GLObject (or a pointer to it) to get the data for that purpose. In the case of this animation assignment, the application was created using the component-oriented paradigm.

***Hierarchical modeling design:***

The hierarchical model was designed in two phases: Load time and Run time. During load time, only the required information is cached, then in run time, this information is used and updated accordingly. Below, a detailed description is given to understand what happens in each phase.

* **Load Time**

In this phase, animation data such as animation stacks, layers, nodes and matrices, is gathered and processed to generate key frames, so that at the end of this phase, only Nodes, their Hierarchy Info and Key Frames are cached. To achieve that, the following classes were designed:

* **MJFBXImporter**: This is a generic fbx import utility, and brings information from an fbx plain text file and decides whether to import to the scene objects that contain animation data or just simple ones. Additionally, it has a pointer to the **AnimationData** class.
* **AnimationData**: This class is responsible to retrieve all the animation information from an fbx plain text file. It does the following:
  + Collects the animation stacks (currently, only one animation stack is used, but the engine is prepared to receive more than one)
  + Sets the default animation layer
  + If a bind pose exists, returns it (for this assignment, one bind pose is assumed)
  + Traverse all the nodes tree to get animation data and stores it in another class named Bone.
* **Bone**: This class is where all the following animation data is cached (if exists).
  + Node name
  + Node VQS transform systems calculated out of the node local transform matrices.
  + Node Key Frames. (\*)
  + Vertex information for a simple cube, to visually represent the node (bone).
  + Parent and Children names.
  + Bone type: Limb or Root

(\*) More importantly, interpolation is computed in this class in order to cache more key frames to generate a smoother animation. This is how it works:

First, the Animation Take is retrieved so that the animation **Timelapse** can be used. From the Timelapse, the **start** and **stop** key frames are stored in temporary variables.

Then, another variable stores the **currentFrameTime** as the start. From that point, a loop iterates over every key frame, extracting their local transform matrix from which the VQS system is generated and pushed into a **KeyFrames\_temp** vector(array). This process is repeated until the current frame time is greater or equal to the stop frame time.

Then, another loop iterates the KeyFrames\_temp vector to generate a final **KeyFrames** vector, by interpolating every two pairs of keyframes’ VQS systems, inserting 4 new VQS systems in between the pair.

Finally, as the last step in the load time phase, the Game Objects are generated using all the information cached in the Bone class. Then, the KeyFrames vector (which has the VQS system for each key frame) is deep-copied in the GLObject class in a vector named **KeyFramesData**. Once all the Game Objects are created, all the animation cache is disposed.

* **Run Time**

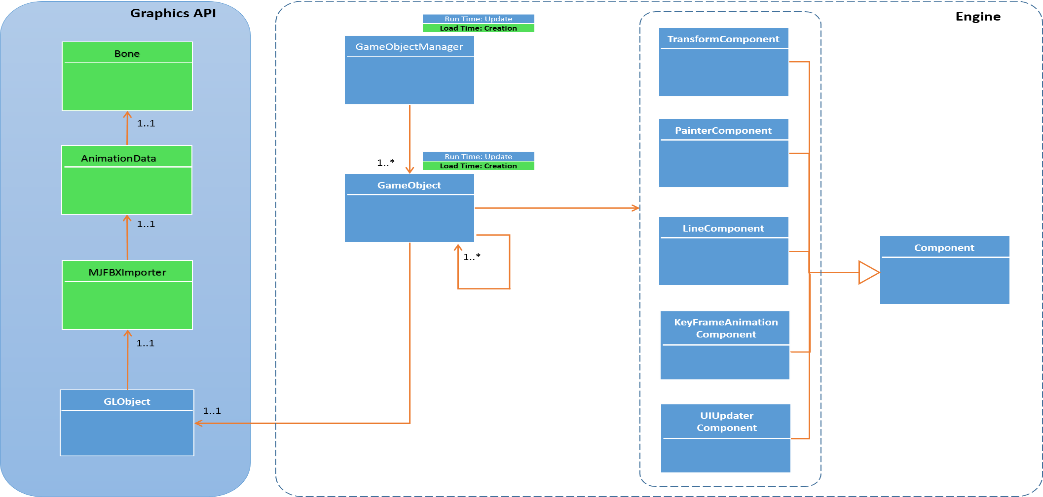
In run time, a **GameObject** class represents a bone, and the set of bones and lines, represent the skeleton that is displayed in the scene. So, in order to implement the hierarchical model, every game object was designed to be a branch of a tree, thus each one has:

* A pointer to its parent **GameObject**. Intended to get parent’s information, such as its transform VQS system (detailed later).
* An array of pointers to children **GameObject**s. This array was intended to get information such as the children count, or to traverse the object tree.

Regarding composition, each bone contains the following components:

* **TransformComponent**: Has a pointer to GLObject class in order to retrieve transformation data. The engine attaches this component by default when a game object is created. That means, all the bones will have their transform data, as well as their parents’, as soon as they are created.
* **PainterComponent**: Has a pointer to the GLObject class, and is in charge of all the rendering preparation and standard drawing procedures.
* **LineComponent**: This has the functionality of drawing lines only if the owner object (parent bone) has a parent. These lines represent the visual link between child and parent bones, and line drawing depends on bones positions.
* **KeyFrameAnimationComponent**: This functionality iterates the **KeyFramesData** vector every application frame, updates the VQS system of the Game Object’s transform and optionally, determines the velocity and smoothness of the animation.
* **UIUpdaterComponent**: **Only the root bone** has this component attached. It was included so that the user has control over the skeleton, that means, the user can move it along the floor plane and rotate it (even at the same time the animation is being reproduced).

Below, a simplified model of the engine is depicted to help visualize classes interaction.



***Hierarchical transformation and concatenation:***

This feature is implemented in the Transform class. There are some pieces of code commented, to show how the transformations used to be computed before (using matrices). Check the Transform.cpp file for more details.

***Utilization of Quaternions and VQS:***

This feature is implemented in the Quaternion and VQS class respectively. Also, VQS is used in load time while extracting the transform matrices from the fbx file. Check out the following classes:

* Quaternion.cpp (whole class)
* VQS.cpp (whole class)
* Transform.cpp (the update function)
* Bone.cpp (the storeTransform and storeKeyFrame functions)

1. **Animation**

***Key frames and motion realism***

The sample fbx provided by the instructor contains a number of 95 key frames which after applying interpolation, key frame number increased up to 4846. With such an amount of key frames, the user can play around with different animation speeds. The engine starts with a speed level of 10, although it can be changed from the UI and can be decreased up to level 1. (Look at the storeKeyFrame Function in the Bone.cpp file).

***Correct animation interpolation methods***

The implemented method was **Slerp**. Check out the following classes:

* Bone.cpp (storeKeyFrame function calls the VQS::Interpolate function)
* VQS.cpp (Interpolate function)
* Quaternion.cpp (Slerp function)

***Smoothness of the animation***

See ***Key frames and motion realism*** for details.