**CS560 – Animation**

**Move Character Along a Path**

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 name 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.

**NOTE**: The documentation below will refer to the location of the classes assuming that the reader will search for them in the Solution Explorer of Visual Studio.

1. **Files of interest for project 2.**

* Spline.cpp
* SplinePathGeneratorComponent.cpp
* KeyFrameAnimationComponent.cpp
* MoveAlongPathComponent.cpp
* RayPlaneIntersectEvaluatorComponent.cpp

1. **Path Interpolation**

***Number of Control Points***

The user interface will let you create up to 8 control points, and the character will start moving when you create the last one. User can create the control points by clicking the middle button of the mouse anywhere in the screen. You can take a look at the **RayPlaneIntersectEvaluatorComponent.cpp** class, located under **SampleApp/Components** filter

***Interpolation of Control Points:***

The space curve was created using a cubic polynomial spline. The function can be found in **Spline.cpp**, located under the **GraphicsAPI/MathUtils** filter. Important functions are commented and provide description of the functionalities implemented.

Also, the visual representation of the space curve is:

* Control points are represented by small green spheres, so you can see them without difficulty.
* Path is represented by a black curve.

***First Order Continuity:***

This is guaranteed, since a cubic polynomial spline have the following characteristics:

* The curve passes through all the control points.
* The curve is continuous and differentiable up to the second derivative.

1. **Arc Length Calculation**

***Arc Length table construction***

The construction of the table is done in the following locations of the code:

* Line #243 of the **Spline.cpp** file, in the function **buildArcLengthTable**
* That function is called in line #195.

***Table concatenation and normalization***

I do the calculation of the elements of the table using one only table, using the advantages of a doubly linked list. This code can be found from lines #282 to 313 in the function **buildArcLengthTable** in **Spline.cpp**.

***Binary search for inverse arc length function***

This implementation can be found at line #342, in the function **UatCurveDistance** in **Spline.cpp**

1. **Speed and Orientation Control**

***Implementation of ease-in/out distance time function***

Two versions of this function can be found in **Spline.cpp**:

* At line #381, **DistanceTimeControlledPeace**, used to get the distance traveled every time t.
* At line #403, **DistanceTime**, used to get the distance traveled at a specific time t.

***Sliding and skidding control***

* This implementation is at line #132 and line #75 in **KeyFrameAnimationComponent.cpp** class.
* It’s worth it to mention that in this same file, I implemented the interpolation in real time, which fixes my approach from project 1. All the implementation is in the **Update** function, and the interpolation is applied at line #160 (You still can find the old code commented)

***Center of Interest for orientation control***

I used an approach based on VQS: I query a point of the space curve, then I generate the W vector. Then, the W vector is normalized and calculate an **angle** between this vector and the global X axis. After that, I create a rotator (quaternion) using the **angle** previously found as the yaw rotation. Finally, send this information to the transform component. You can find this implementation in the Update function of the **MoveAlongPathComponent.cpp** class at line #78

1. **Visualization**

Required features are implemented in a window and instructions on how to use the camera and UI in general is described in the UI itself. Full screen is not supported.

1. **Documentation**

Required code is appropriately commented. Instructions about where to find features implemented and their comments are in the previous sections of this document.

1. **Extra credits**

N.A.