**Technical Animation Final Project:**

**Spatial Key framing for Performance-driven Animation**

Ruchi Hendre

1. **Problem Statement / Goals**

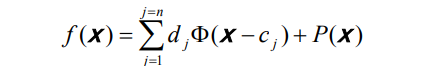
The goal of this project is to create a toned down version of the 2005 paper, Igarashi’s Spatial Key framing for Performance-driven Animation. This paper introduces spatial key framing, a technique for performance-driven character animation. This is a technique which created animations from scratch and does not need any motion capture data for implementation.

1. **Method**

Spatial Key frames for the system are defined as a cursor position associated with that spatial keyframe and also the associated pose that is a 3\*3 matrix.

This paper uses Radial Basis Function for interpolation. For interpolation between two points each entry of the rotation matrix is interpolated using a radial basis function and the resulting matrix is then orthonormalized.

The Radial Basis Function is defined as follows it is very useful for scattered data interpolation, the basis function for this system as directed by the paper is the |x| function.



For each entry in 3\* 3 matrix solve for dj such that f(x) = given pose at Cx,Cy,Cz (marker locations)

Cj, - Marker positions.

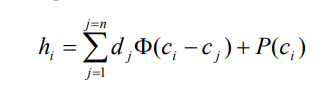
Dj- weights

P(X) – degree one polynomial.

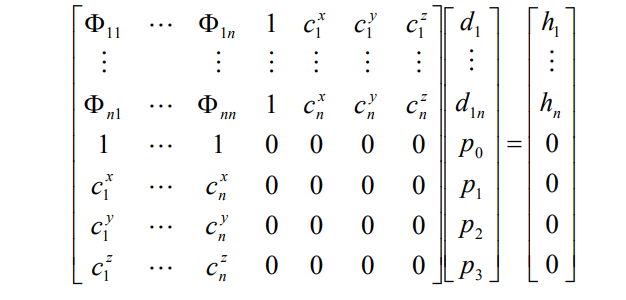
F(x) - radial function ( |X| in the paper)

For root part – translation motion

Calculate F(X) for the translation vector using this interpolation

****

Interpolation system relies on the assumption that the interpolation function f(x) can be found through a linear combination of basis functions. So our system needs to satisfy the equation f(xi) = hi, this equation gives rise to the following linear system, first we need to find the weights d1……dn and the linear term, these values will give us the interpolation system.



We obtain the interpolation function f(x) by solving the above linear system. We need to solve the linear system for each entry of the 3×3 rotation matrix. So we will get 9 interpolation function in total which will give us the transformation for our joint which will be included in a 3x3 interpolation matrix.

This matrix that we get needs to be orthonormalized as it is not a orthonormal rotation matrix, the method for orthonormalization outlined in the paper is that of iterative refinement.

This orthonormalized matrix is then applied to the concerned joint.

1. **Implementation.**

For this project I used Unity 3D and a 3D human model which was rigged using Maya’s Auto Rig Feature.

**Step 1: Importing a fully rigged 3D model into Unity.**



I imported this model in Unity 3D, and used a spherical collider where the figure joints were with which I was able to rotate the joints of the model during runtime. This model was modeled by me on an earlier semester.

**Step 2: Creating Spatial Key frames.**

In the final deliverable of the paper there is a mechanism for creating spatial key frames at runtime, I tried to replicate this functionality by having a translatable cube as my cursor, and this cursor sets a spatial key frame (sphere) when the set spatial key frame button is clicked.

It saves the cursor position and the pose to a text file SpatialKeyFrame.txt which has the following structure.

[0.03306098, 0.6888634, 0.7193463, 0.08315936]

Cursor[1.083972, 0.7394111, -9.309]

[0.5796403, 0.2554487, 0.3599544, -0.6849789]

Cursor[0.241819, 0.7230731, -9.309]

I am saving a quaternion here but converting it into a rotation matrix using Unity inbuilt Quaternion Library.

**Step 3: Interpolating between spatial key frames.**

The next step was interpolating between spatial key frames I tried out the Algorithm for 1D first, this algorithm worked fine after comparing with the solved examples.

* 1D Interpolation for 2 points.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **1. Cursors: ( 4, 0 , 0) , (12,0,0)** | Theta: 30 degrees and 90 degrees. | | Coefficient Matrix:  0 8 1 4  8 0 1 12  1 1 0 0  4 12 0 0 | Result Matrix:  0  0  0  7.5 | | Interpolation Function:  7.5x |
| Cursor Position | | Sum (From Program) | | | Sum: 7.5X | |
| (5.0, 0.7, -1.6) | | 37.23992 | | | 7.5 \* 5.0 = 37.5 | |
| (9.9, 0.2, -1.6) | | 74.58514 | | | 7.5 \* 9.9 = 74.25 | |
| **2. Cursors: (2,0,0) , (10,0,0)** | Theta: 30 degrees and 90 degrees. | | Coefficient Matrix:  0 8 1 2  8 0 1 10  1 1 0 0  2 10 0 0 | Result Matrix:  0  0  15  7.5 | | Interpolation Function:  15 + 7.5x |

Math: The worked out example for the second example is given within this folder based on solution shared with me.

The code for 1d interpolation and checking if it works fine is included in the code for InterpolationScript.cs.

* 1D Interpolation for 3 points: Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **1. Cursors: ( 2, 0 , 0) , (6,0,0) ,(12,0,0)** | Theta: 60 degrees, 0 degrees and 90 degrees. | | Coefficient Matrix:  0 4 10 1 2  4 0 6 1 6  10 6 0 1 12  1 1 1 0 0  2 6 12 0 0 | Result Matrix:  -9  15  -6  54  3 | | Interpolation Function:  -9 \* (|x-c1|) + 15(|x-c2|) -6 (|x-c3|) + 54 + 3\* x |
| Cursor Position | | Sum | | | Result | |
| (4, 0 , 0) | | 30 | | | Correct | |
| (6, 0 ,0) | | 0 | | | Correct | |

Math: The worked out example for the second example is given within this folder based on solution shared with me.

* Algorithm for 3D Rotations.

As mentioned in the paper the procedure for the 3D rotations is very similar to the above 1D Examples, only difference is that using the 3\*3 rotation matrices instead of giving the angles around the axis. The procedure followed is as mentioned in the paper itself except the orthonormalization process for which Unity.Vector3.orthonormalize is used.

Some Numerical Results from the program:

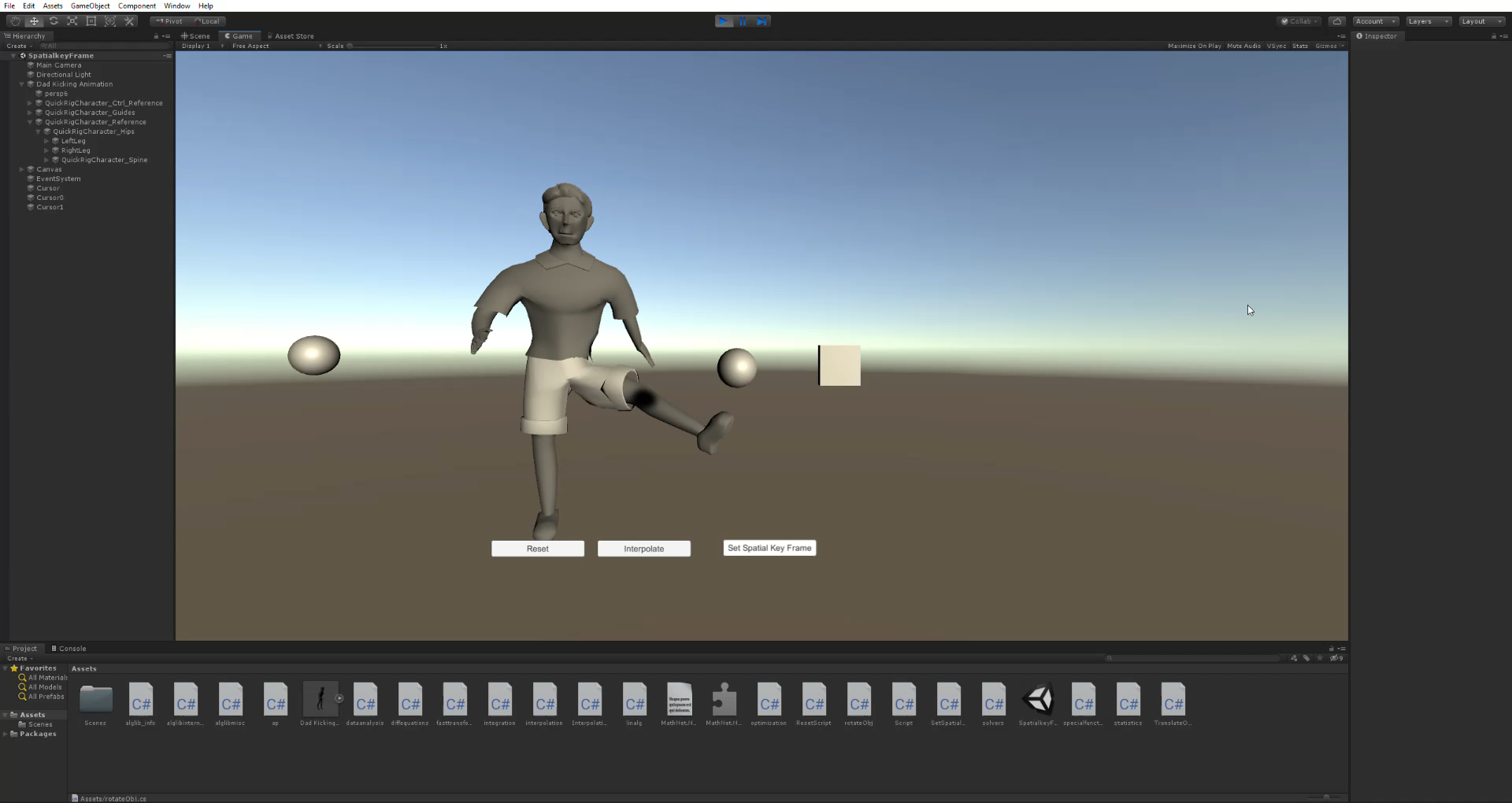
|  |  |  |  |
| --- | --- | --- | --- |
| 1. **Cursors** (2, 1, -5) and (6,1, -5) | **Thetha:** Poses which were saved as quaternions  firstPose.X = 0.626317f;  firstPose.Y= 0.1404756f;  firstPose.Z = 0.7152255f;  firstPose.W = 0.2764893f;  secondPose.X = 0.5487222f;  secondPose.Y = 0.1102552f;  secondPose.Z = 0.6743674f;  secondPose.W= 0.4816397f; | **Coefficient Matrix:**  0 4 1 2 1 -5  4 0 1 6 1 5  1 1 0 0 0 0  2 6 0 0 0 0  1 1 0 0 0 0  -5 -5 0 0 0 0  **HMatrix for first entry of rotation matrix**:  -0.0625619  0.0661447  0  0  0  0  **DMatrix calculated for this first entry:**  5.64936E-17  -2.64428E-17  0.00286048  0.0493771  0.00286048  -0.0143024 | **Interpolated Matrix:**  for Cursor (5.0, 0.7, -4.2)  0.05261426 0.6456767 0.5398709  0.4385724 0.439544 0.5686133  0.7177942 0.3148569 0.310406  This interpolated matrix is orthonormalized using Unity.Vector3.orthonormalize and applied to the joint.  **Orthonormalized Matrix:**  0.06239213 0.7656698 0.6402009  0.5209146 -0.522167 0.67527  0.8513256 -0.3756215 0.3662695 |

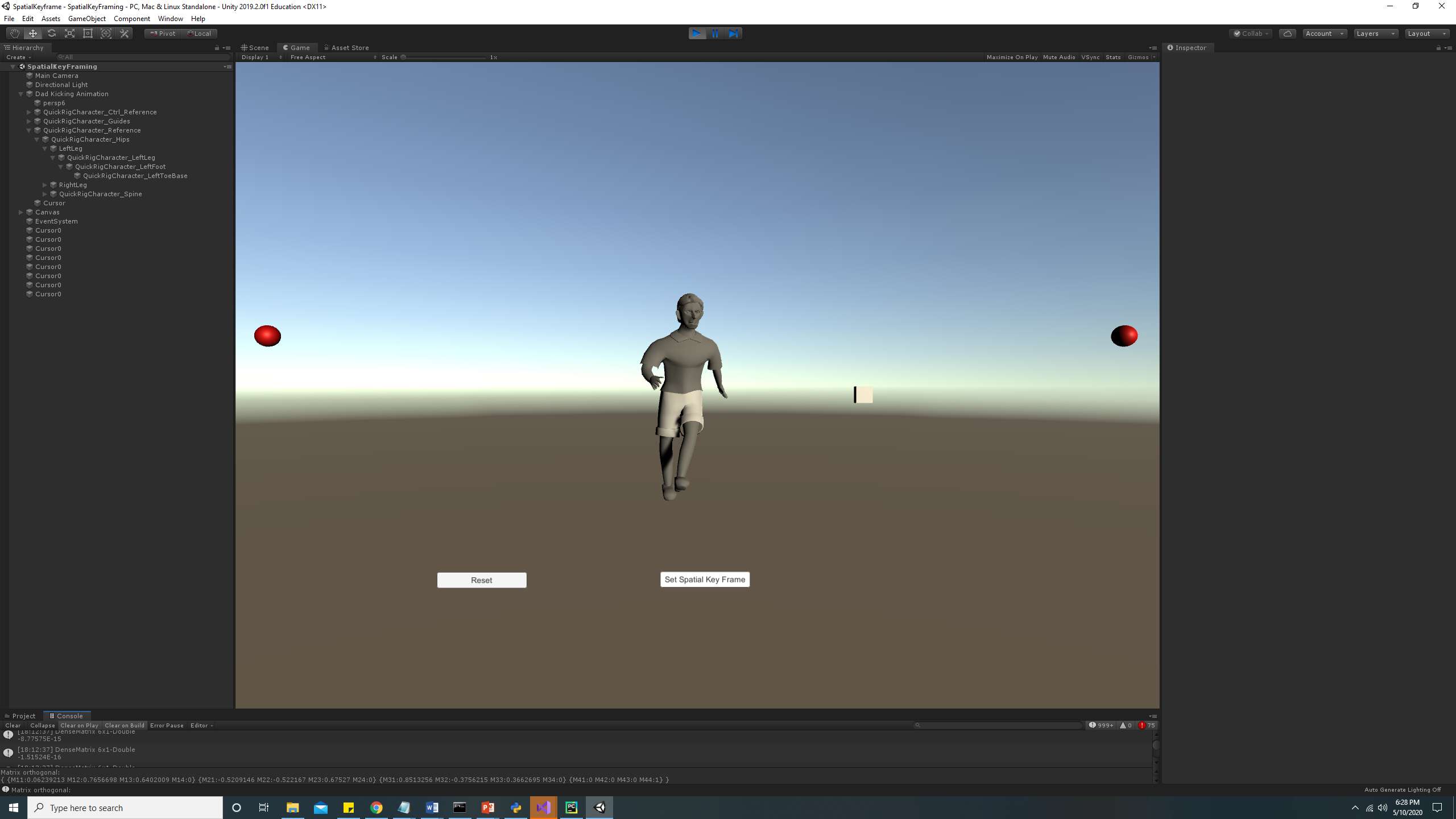
These results are obtained from the program and may require further tweaking to the algorithm, currently I am using an inbuilt solver within the MathNet.Numerics library.

I believe the jittering effects and not meeting the correct keypose for the spatial key frame is happening because of using less number of keyframes and the above iterative solver.

Demos: The Demo Videos for this example are attached within this folder at the location **Final Project Report\Demos**

Screenshots: Here are some screenshots from the system





1. **Problems faced**

* The biggest problem I faced was in understanding the algorithm for and how it works with respect to a 3\*3 matrix, and what would be a h(i) for a 3\*3 matrix. If I had more time I would have liked to understand how the coefficient matrix is formed, and what is the understanding behind the radial basis function.
* Since I am new to Unity it took me a while to understand the functions and the apis for the calculations required for the implementation, I did find libraries that aglib and also spent time looking at other options in the numpy class in python. Unity gave the best interface for manipulating and creating keyframes hece went forward with unity.
* As it is mentioned in the paper when the spatial key frames are less than 4 and they are linearly or two dimensionally distributed. There are inconsistencies with respect to the interpolation mostly concerning scale. This could be due to the code itself or due to using inbuilt functions to convert quaternions to rotation matrices.
* Singularity and sparseness of the coefficient matrix for two and three spatial key frames make the linear solver result doubtful.

1. **Resources**

Unity Documentation for Orthonormalization: <https://docs.unity3d.com/ScriptReference/Vector3.OrthoNormalize.html>

MathNet for linear algebra operations: <https://numerics.mathdotnet.com/>

System.Numerics.Quaternion for Quaternion to Rotation Matrix conversion.

Papers:

Spatial Keyframing for Performance-driven Animation T. Igarashi 1,3, T. Moscovich 2, and J. F. Hughes 2 <https://www-ui.is.s.u-tokyo.ac.jp/~takeo/papers/squirrel.pdf>

Modelling with Implicit Surfaces that Interpolate GREG TURK Georgia Institute of Technology JAMES F. O’BRIEN University of California, Berkeley<http://graphics.berkeley.edu/papers/Turk-MIS-2002-10/Turk-MIS-2002-10.pdf>