GDW Justification Report

Ken Ho – 100583602

Jayden Cooper – 100582362

Simon Pichl – 100579423

Ray Dee – 100578667

Darius Hackney – 100584623

Cameron van Velzen - 100591663

GiT GüD Games

UOIT |

GDW Overview – GiT GüD Games

The game:

A 3D stealth puzzle where you must use the shadows to return to where you rightfully belong.

The game that we have been making this semester is, as the one liner entails, a three dimensional puzzle game where you play as the shadow of a child. The plot points of the game revolve around the player playing as the shadow in each level of this child's house. The player must make their way through the levels of the house to find their missing child again. Each level is a different room in the house that the player will progress through. The reason we use the term stealth puzzle is because the aura of the game is dark and mysterious. There are no enemies in the game in the sense that there are no other player-based characters that the protagonist can run into. What we refer to as "enemies" in our game are the lights that can be found in each level. The object of each level is to avoid the light because if you touch it you will die and lose a life, which you only have three. The player must maneuver their way around static and moving lights to make it to the completion of the level. Each room of the house will display different types of challenges but still using similar assets that the player recognizes. For example, there will be lights that swing back and forth in a line and those lights will be seen in almost every level but each one will have a different obstacle added to it like another light in the middle or a static light that it passes through making the player think about the puzzle differently.

The Team:

GiT GüD Games is made up of six members; Jayden Cooper, Raymond Dee, Darius Hackney, Ken Ho, Simon Pichl, and Cameron van Velzen. Our team roles are as follow; Designers: Cameron & Darius, Artists: Jay & Ken, Programmers: Ray & Simon. These roles simply represent our strengths amongst the team.

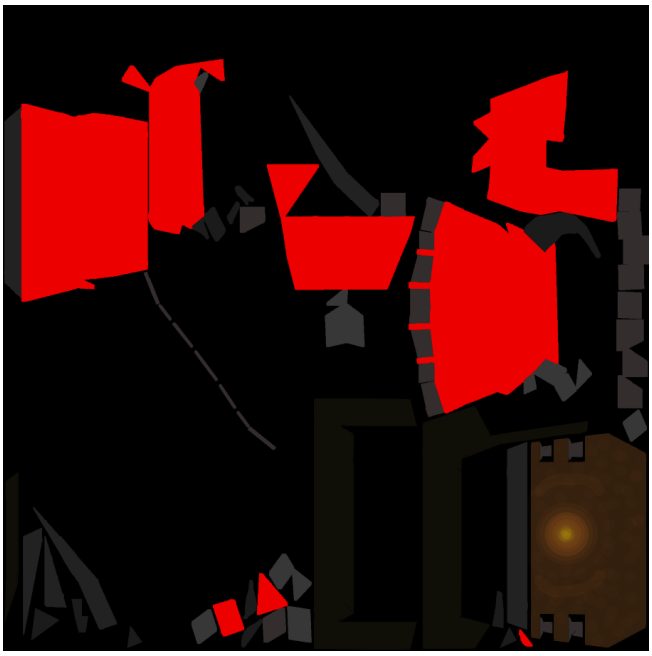
Progress Report:

Because our team is divided up very nicely, we have an even balance of the three core practices used in making games, in our opinion, being Art, Design, and Programming.

Art – Jay & Ken:

From the very beginning of making the game, after we have all brainstormed roughly what we want to expect, the artists start to think about the style. Based on the rough concept, they will make a style and theme that the brain will associate with what the players will be seeing as they play. Once they have decided upon a general style they will use to create all the objects and textures in the game they will then create colours to appeal to their specific style set. This is what will make

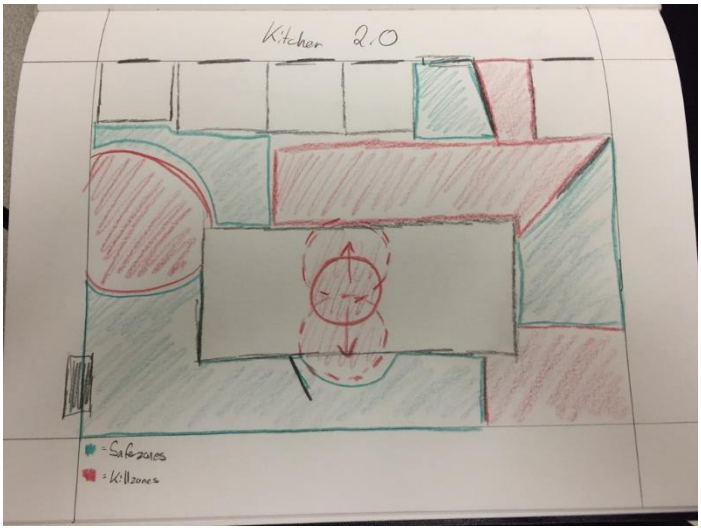
the gameplay come alive as the players' progress through each level. First the artists will be asked for basic concept art and models for testing purposes. Say for example the stairs. Next they will create the barebones of what stairs might look like, simple rectangular looking stair case. This will allow them to gain the rough idea of what size all of the models will be. Once one object works they will test it next to another object as well as the player, to see if the scale of all the objects appeals to the mind. This will apply to the entire gameworld as well because the objects will be somewhat to scale with the room as a whole. Once the placeholder objects have been approved and deemed functional, the artists can make them more complex. This includes stuff like adding a railing to the stairs, hinges and a door to the fridge etc. and polishing the models to match the style and design of the game as intended. Once every designed object has been turned into a model and fixed up, they are ready to be fully textured. The models are UV unwrapped and properly textured to be loaded into the game just as they looked before but now with textures. The tricky part about adding the textures to the game is colouring them differently than you would think. The artists must texture the objects as if there was a high intensity light being shinned on the object from all angles, because the shades of these colours will be implemented with the shaders in the game code. Finally, from the artists, they will work with the designers to create the states and scenes used for HUD and UI elements as well as the main menu and such. Once the designers are satisfied with the correct use of each element, the artists will create assets to be put in the spots of those designs, being life counters and telling the player what to press if they win or lose.



Design – Cameron & Darius:

Brainstorming and deciding on the idea and scope of the game is a group effort because everyone has opinions and ideas to contribute to the overall making of the game. Designing comes into play in the ability to create the gameplay of the game. We are tasked to make the game play how the player thinks it would be. Now what does that mean? Programming obviously makes the literal framework of gameplay, but as designers, we take into account the player actually playing the game. We are in charge of creating a fun and fair game to the target market we are aiming to. Designing starts with the core gameplay loop along with the core mechanics of the game, then all the juiciness comes in after. First of all, we have to start with an idea/a genre of game we want to make. Deciding on puzzle games, this allowed us to narrow our thoughts onto the type of gameplay we will have; that being a thinking challenge for the player. Because we decided on not having and A.I. or enemies, there is not skilled gameplay that the player must pay attention to, but rather mind games that you will play with the player, forcing them to think about each problem, as well

as letting them learn from their failures or successes. Along with the idea of the game, we decided to include multiple levels that the player can go through, because this fits with the theme of our setting, being in a house. Now each level is designed independently of each other, using the same or similar assets from previous levels. In the basement level there is a fridge that produces a light for the player to avoid. In the kitchen there is a fridge as well (obviously), but in a different place, allowing the player to still see something familiar but in a different way. This challenges their brain to adapt to new situations in every level. The levels are designed on paper first, as paper prototypes, so they can be adjusted and played without the need for actual game play yet. Once tested on paper and the artists and programmers have implemented base models, we can then start to implement the puzzles that each level has on paper. This allows more peers to experience the design and possibly find things wrong with the design, like maybe its too hard or too easy which doesn't make the game as fun as it could be. The designers want to achieve a sense of flow in the player where its not too easy that they get bored but its not too hard that they quit. This will allow the last step of the level design to be completed which is playtesting. Getting unbiased persons to play the game and record data while they play and ask specific questions each time. This allows us to gather a strong data structure of ideas or feedback on each design which will make it better for the player. Once the levels have been structured properly, the final step to integrating with the player through the game is the UI and HUD elements. We gather positive feedback for good elements of HUD design as well as reference other games to reinforce our choices. This is important because this is the only interaction that the player has with gameplay and the feedback they receive depending on what choices they make.



Programming – Ray & Simon:

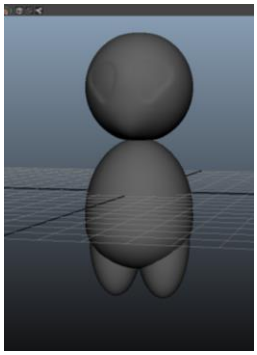
This is what really makes everything come to life. The code allows us to manipulate everything we have in our heads and transfer is into an actual game. This is where the game world is created and implemented for the payer to experience everything we have thought up before. Although everything is implemented here, you cannot just jump right to it. For everything to work the way it does, first and foremost there needs to be a framework of the essentials of the game, most importantly coming down to loading objects. Creating the object loader allows us to implement and test things as soon as we need to so we can make them better faster. Once there is object loading in the game world; the game world is the framework we have set up and implemented a camera for the player (and us) to see exactly what we are making. Once we know where something is in the world and have an origin point where every object is loaded, we can modify the positions and orientations of all those objects using formulas set up in the libraries used for math in the code. These libraries contain the necessary data used to allow these formulas to work the way they are being implemented. For example, to modify the position or rotation of an object we use the respective matrix and apply it to the transformation of the position vector of the object we are using. This lets us place each objects exactly where we want and expect them to be in the level. All of the libraries and files we create are used to call upon multiple functions to access what we want. What this means is everything has shared characteristics which allow us to call them all as a whole. We have game objects which are our scene objects as well as a game object that is our player character. This makes it so we can apply different transformations and functions to the player separately, because logically we wont be giving a couch player controls. Once each object is loaded, textured and placed properly, other players will now be able to complete the gameplay loop and find potential bugs or issues for feedback so we can make the game more polished and enjoyable. This includes things like missing or offset collision boxes, errors that occur between levels or if things are not placed where we thought they were. Along with everyone else, finally there comes the interaction with the player. Programming the main menu, the win, and the game over screen along with the HUD and UI keeps the player integrated with the game instead of breaking their connection and immersion the playability of the game.

```
glm::mat4 Transform::translate(const float x, const float y, const float z)
{
    glm::mat4 moveyMatrix(
        1.0f, 0.0f, 0.0f, x,
        0.0f, 1.0f, 0.0f, y,
        0.0f, 0.0f, 1.0f, z,
        0.0f, 0.0f, 0.0f, 1.0f);

    return matrix = (matrix * moveyMatrix);
}
```

```
float * Transform::getFirstElement()
{
    return &matrix[0][0];
}

glm::mat4 Transform::rotateX(const float theta)
{
    float cosT = cosf(theta);
    float sinT = sinf(theta);

    glm::mat4 rotateyMatrix(
        1.0f, 0.0f, 0.0f, 0.0f,
        0.0f, cosT, -sinT, 0.0f,
        0.0f, sinT, cosT, 0.0f,
        0.0f, 0.0f, 0.0f, 1.0f);

    matrix = (matrix * rotateyMatrix);
    return matrix;
}
```

Algorithms & Contributions:

Jayden Cooper – Artist



Creating the character model is one thing because you can add player controls to a character like figure, but you can also do the same to anything; like a couch. So the important part of having the character is rigging it so that animations can be applied to it, to give it realistic motion that the player would expect. This also allows us to give the player feedback constantly, because it reinforces the direction and just the fact that they are walking. Having the character model rigged and skinned, this allows us to apply animations to the model. Making simple animations frames of the model for it to complete a walk cycle.

Ray Dee – Programmer

This is where the true implementation of the algorithms is made. The math and animation headers and libraries allow us to input the formulas necessary to accomplish the correct tasks. What we mean by this is specific algorithms such as linear interpolation and curves. These formulas allow us to create moving (LERPing) lights that the player has to avoid as they progress. The curves are used in a similar fashion except instead of the lights moving in a horizontal path, the lights move in a curve or a path that the player might not expect the first time around. These formulas can be used in conjunction with each other to make unique combinations of moving lights (LERP and curves).

```
sceneObjects["SquareLight"]->setPosition(lerp(glm::vec3(12.0f, 0.1f, 25.0f), glm::vec3(20.0f, 0.1f, 25.0f), ((cosf(totalTime) + 1.0f) * 0.5)));
sceneObjects["SquareLight2"]->setPosition(spline(glm::vec3(50.0f, 0.1f, -17.0f), glm::vec3(7.0f, 0.1f, -17.0f), glm::vec3(-10.0f, 0.1f, 0.0f), glm::vec3(-10.0f, 0.1f, 50.0f), ((cosf(totalTime) + 1.0f) * 0.5)));
sceneObjects["SquareLight3"]->setPosition(lerp(glm::vec3(12.0f, 0.1f, 0.0f), glm::vec3(27.0f, 0.1f, 0.0f), ((sinf(totalTime) + 1.0f) * 0.5)));
sceneObjects["Lamp"]->setPosition(lerp(glm::vec3(14.0f, 35.0f, 25.0f), glm::vec3(17.0f, 35.0f, 25.0f), ((cosf(totalTime) + 1.0f) * 0.5)));
sceneObjects["Lamp2"]->setPosition(spline(glm::vec3(50.0f, 35.0f, -17.0f), glm::vec3(7.0f, 35.0f, -17.0f), glm::vec3(-10.0f, 35.0f, 0.0f), glm::vec3(-10.0f, 35.0f, 50.0f), ((cosf(totalTime) + 1.0f) * 0.5)));
sceneObjects["Lamp3"]->setPosition(lerp(glm::vec3(14.0f, 35.0f, 0.0f), glm::vec3(20.0f, 35.0f, 0.0f), ((sinf(totalTime) + 1.0f) * 0.5)));
```

Darius Hackney – Designer

Designing a level will come before programming all the pieces in that level, so doing so, we have to take into account what we can and can't do. Basic physics were a given and this allowed us to provide unique ways to think your way around puzzles, by pushing and pulling objects in the way of the light. Keeping the scope of these topics simple for both parties allows us to transfer paper data into code, more importantly mathematical equations that can represent basic Newtonian physics to apply forces to objects. Another aspect of the algorithms component is the design of the HUD which uses sprite-based animation to notify the player how many lives they have left because they get a game over.

Ken Ho – Artist

Although it may not seem like the art applies to the algorithms of the game but its actually what makes them truly come to life. Making objects and lights that are able to follow a path and LERP the way they do so that the player can see the danger that they face. This coincides with the constant feedback that we provide to the player; that the light will kill them. The static lights do not apply to algorithms, but the moving ones do. These have to be a specific type of light, small enough that its fair and not too hard to get around, but not too small that its not a challenge as it

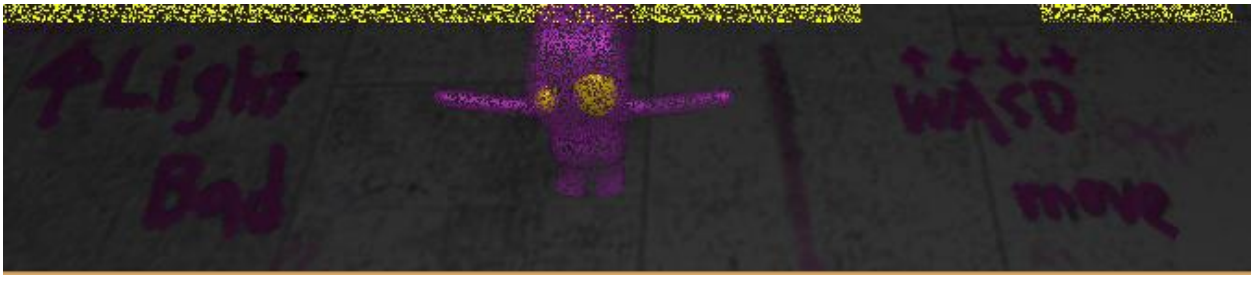
moves back and forth. Keeping in mind the LERP or curve path that this light will follow is what creates the shape and size of the necessary light that will be moving. Another key piece of the algorithms is the sprite based animations. These were created in a sprite sheet to be animated through the game in the UI or HUD to make the game feel more alive and again; provide constant feedback to the player.

Simon Pichl – Programmer

Implementation of a model is as simple as loading objects because the model itself is just another object file, but getting the model to move how anticipated is a different thing all together. Applying animations to the character reinforces the gameplay and feel of the whole experience of the game. Once there is a rigged model, it can be coded to morph and lerp through its target animations that were planned for it. Having movement is one thing like making "sphere-man in box-world' because movement can be applied to anything we want, but making the sphere come to life with the appeal of the environment around it. Making the joints of the model morph so that the animations look more "life-like" and making each corresponding joint and vertex LERP to the new position to create multiple frame of animation that will make up the character's cycles, such as walk and idle.

Cameron van Velzen – Designer

Designing each level individually keeping the flow of the game as fun as possible is a challenge in itself, but the more important thing is deeper than the designing of the puzzles, it comes in the equations and algorithms needed to make those puzzles come to life. LERPing is essentially one of the most important algorithms necessary for the game because it is very versatile in making multiple things behave the way they are intended. With the correct algorithms in mind, designing these puzzles becomes easier for all parties because making the light move the way that we intend is a matter of plotting the coordinates necessary given the parameters of the equations. Making curves is the same idea, but more in-depth because of the non-linear sense of points. Planning the points accordingly such as the start and end point, but also the points in the middle that modify the path of the object from beginning to end.

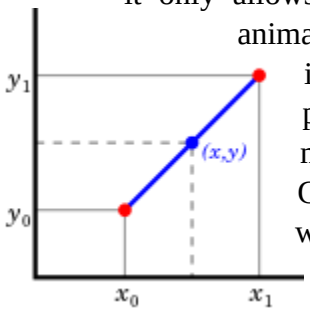


Animation Algorithms & Implementation

LERP

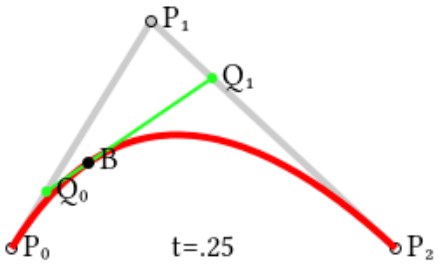
Linear Interpolation, or LERP, in math is a method of estimating data that may be unknown or un listed in a data set. In algorithms we would use it to move objects between two or more points. This is a very fundamental algorithm necessary and applicable to be used in multiple different ways within the make of our game. Linear interpolation makes the assumptions that you are aware of the end points that you wish to interpolate between in a straight line, unlike spline interpolation which connects the points in curved lines. Mathematically this algorithm works between two points, so the formula will accept the start and end point, then it will accept the desired variable "*t*" which represents the time constraint you wish to apply to the LERPing target. This concept is easy for us to implement and use in the game but it is very basic; it only allows for straight path movement which doesn't let use move smoothly between animations. This doesn't mean that it becomes useless to us. LERP is not primarily used in the motions of characters, but it is used in the movement of our most basic light puzzles. Using the formula, we can set up two points in space in the game world and make the object we chose to move from one point to another or back and forth. Giving the start and end points makes it so the object knows the path to follow, but we also pass it the value of perceived time in the game so that it gives the illusion of a swinging light hanging from the ceiling. Adding multiple lights with the same formula allows us to not only move the points around the game world, but also modify the given time values using sine and cosine functions being manipulated so that the lights are offset from each other. Each floor light is a rectangular shape and the hanging ceiling light is a cone-like shape that swings with it. An interesting feature about the way we use LERP with similar objects is the floor light and its corresponding ceiling light. If we make the ceiling light's LERP path to be inside the range of the floor light's LERP path, so that when the lamp reaches the edge it's path and the floor light continues, it forces the brain to think that the lamp itself is swaying back and forth on a tilt like a real hanging light would do if it were to swing. Because our LERPing lights are all modified by our perceived time in the game, they obey a natural swinging effect that one would think would occur in real life because given that the game is running at optimal time; 60 frames per second, the lights will be swinging over time at the desired speed.

```
static glm::vec3 lerp(glm::vec3 P0, glm::vec3 P1, float t)
{
    return ((1.0f - t)*P0 + t*P1);
}
```



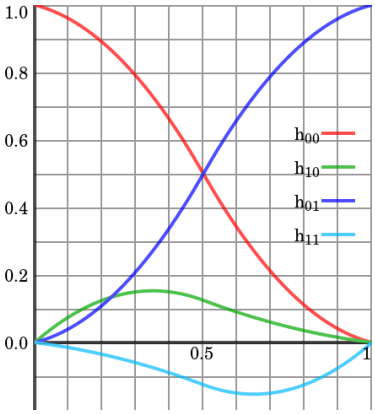
Curves

Using splines becomes more efficient to us when we are moving between points in space. A spine is simply just a curve that is defined by specific points along the curve that modify its path. These are known as control points. Much like linear interpolation, we are moving between points, but not just two. Splines allow us to take two points and depending on the points between them or before and after, will affect the path of object moving on the given path. Mathematically, given a data set of



points, and outlining specific parameters in the equation, the formula will take the points between your two desired points and use those as “control points” for the curve. This means that they will modify the path that the object will follow instead of going direct in a straight line. This becomes advantageous to us when we want to move objects in a less slug-ish way and make them smoother. Instead of going directly to the other point, a smooth path is achieved between them. More specifically this applies to Bezier splines. Creating a curve using Bezier will morph the path in the direction of the modifiers until the end point is reached. Bezier specifically in our game we have quadratic splines.

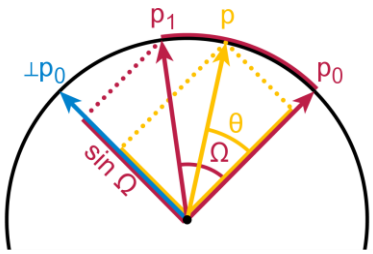
```
static glm::vec3 bezier(glm::vec3 P0, glm::vec3 P1, glm::vec3 P2, float t)
{
    return ((1 - t) * ((1 - t) * P0 + t * P1) + t * ((1 - t) * P1 + t * P2));
}
```



What this means is we have three points within the parameters of our Bezier equation and those three points correspond to a point in three dimensional space where the curve is applied to. The fist point being the start of the curve, and the last point being then end. It is the middle point that acts as the modifier and wherever it is placed the path will be affected by it, very similar to a magnet in the sense that the path will not travel straight through each point, but will be pulled towards the control point as it goes from beginning to end. This allows us to make more diverse moving lights by having them travel in a radius like path to and from the points that we set it up to be. These can be used in conjunction with the LERPing lights mentioned above to create more difficult challenges for the player to avoid, especially when they are desynchronized, as mentioned in the sine and cosine functions before. The second type of splines used are similar but more advanced being cubic splines, specifically Catmull-Rom. Now the large difference in these curves is that our Catmull-Rom spline is cubic, meaning that it belongs to the basic hermite curve set. These splines have two end points and two control points between them, allowing for more intricate curves and paths. Each point can be modified independently of each other allowing multiple different combinations of the same equation in use. Another difference between the two is Catmull-Rom will take the points before and after it specifically to align the path according to the data it has been given. This will allow a smoother path between the keyframes of each animation of the character, making him feel more life-like and not clunky to control. These types of curves make it so the path arcs in weird or unique ways that the player might not expect, such as a curved 90-degree angle or an “S’-like motion.

```
static glm::vec3 spline(glm::vec3 P0, glm::vec3 P1, glm::vec3 P2, glm::vec3 P3, float t)
{
    return 0.5f * ((2.0f * P1) +
        (-P0 + P2) * t +
        (2.0f * P0 - 5.0f * P1 + 4.0f * P2 - P3) * (t*t) +
        (-P0 + 3.0f * P1 - 3.0f * P2 + P3) * (t*t*t)
        );
}
```

Spherical Linear Interpolation

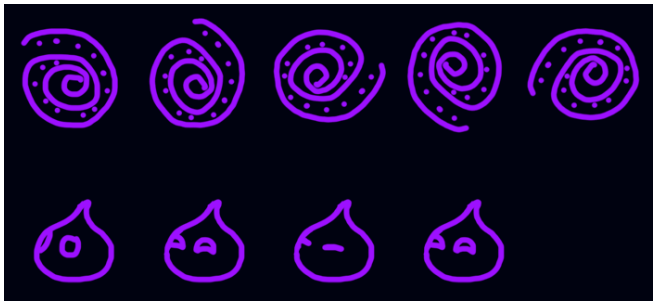


This concept is similar to straight line linear interpolation except it is used more precisely. Spherical Linear Interpolation, or SLERP, refers to the constant movement along a radius given the end points and parameters for the interpolation. Much like LERP, SLERP still uses beginning and end points, but imagine if you took the straight line of a linear interpolation and bent the end point to an even radius around to the start point. This will create the path required to follow but in an even radius. The use and implementation of it applies to the rotation of object, specifically player controlled object. Making the player look in the direction that the player is telling it to walk is a matter of rotating the model to look forward, back, left, and right but adding SLERP will make the character model interpolate from its beginning start position to its desired end position in a radius, in this case being itself. Logically it will have points in each direction, similar to cardinal points, and will SLERP between them when given the proper command on where to look.

```
static glm::vec3 slerp(glm::vec3 P0, glm::vec3 P1, float t)
{
    float cosOmega, scale0, scale1;
    cosOmega = glm::dot(P0, P1);
    if (1.0f - cosOmega > 1e-6)
    {
        float omega, sinOmega, oneOverSinOmega;
        omega = acos(cosOmega);
        sinOmega = sin(omega);
        oneOverSinOmega = 1.0f / sinOmega;
        scale0 = sin((1.0f - t) * omega) * oneOverSinOmega;
        scale1 = sin(t * omega) * oneOverSinOmega;
    }
    else
    {
        scale0 = 1.0f - t;
        scale1 = t;
    }

    return P0 * scale0 + P1 * scale1;
}
```

Sprite-Based Animations

Primarily used in our two dimensional games, making a sprite move between its animations makes them come to life. These include things like walk or idle animations, or even decaying health bars or resource bars (mana, stamina etc). Having sprites is an easy way to continually integrate with the players' experience because it lets them track their own progress. A sprite sheet is made with each different sprite in each different planned frame and when it is implemented it is passed through in the right order to make it appear correctly at the right time. Our main character will have 4 lives, resetting every level, and when those are depleted they will get a game over. To tell the player about this current status there will be a sprite sheet that contains the number of lives the player has, and it will flip through them as necessary if the player dies; they will lose one, or if they complete the level; they will be back at full. Challenges with this come up when mixing the two dimensional sprites with the three dimensional game world and applying it to the HUD and foreground for the player to always be able to see. These sprites have to be the right size and orientation for the maximum amount of information to reach the player without distracting them. This will also be used in menu and UI



aspects to add life to the game screens, such as moving sprites in the corners to give more appeal to what would have thought to just be a bland screen.

Basic Newtonian Physics

```
Piano.BBBL = glm::vec2(-2.8f, 6.9f);
Piano.BBFR = glm::vec2(3.0f, -6.9f);

LoveSeat.BBBL = glm::vec2(-2.8f, 6.9f);
LoveSeat.BBFR = glm::vec2(3.0f, -6.9f);

Table.BBBL = glm::vec2(-2.8f, 6.9f);
Table.BBFR = glm::vec2(3.0f, -6.9f);

Chair.BBBL = glm::vec2(-2.0f, 2.0f);
Chair.BBFR = glm::vec2(2.0f, -2.0f);
```

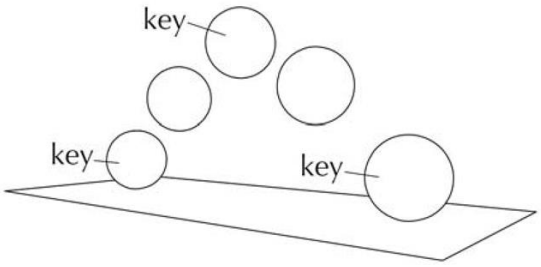
For a game to feel as realistic as possible, it should have as much relation to the real world as possible so the player can experience similar aspects that the brain is familiar with. The easiest way to achieve this feat is through the use of basic physics. What this means is on a daily basis there are things the brain knows to be true and these are simply the laws of the universe. Things like gravity and motion are of the simplest concepts that the player doesn't necessarily know they are thinking about it but they just already know it. With gravity there are things such as forces that are inherently accompanied with it. This is because gravity is constantly applying a force downwards to everything around it. Without too much detail, essentially the heavier things are, the harder they are to move. Our character is made out to be the shadow of a little child and in the game world there are objects that you would typically find in a house. Things like a couch or a fridge. Physics allows us to move into those object and hit them, because we are applying a force to it, but it is also applying an equal force on us, so it won't move. Only when our force exceeds the opposing force will we be able to move that object. This is where our game uses these simple laws in the movement of objects so the player can maneuver through levels differently. This is by means of pushing or pulling objects that are light enough to move in the way of lights to create a new, safe path for yourself. Advantages to this are that we can make more in-depth puzzles with multiple features to challenge the player once they feel their skill is high enough. A problem is that the player may begin to think that every object can be move or can protect them, giving them too much false hope. Also this can lead to puzzle that provide too much challenge to a player, which if combined with the previous statement, would lead to frustration and experience-breaking gameplay.

```
if (input.GetKey(KeyCode::D) && colliding == false)
{
    //setPosition(glm::vec3(getPosition().x + 0.1f, getPos
    velocity.x = appliedVelocity;

    transform.zeroMatrix();
    transform.rotateY(1.57079f);
}

if (input.GetKey(KeyCode::A) && colliding == false)
{
    //setPosition(glm::vec3(getPosition().x - 0.1f, getPos
    velocity.x = -appliedVelocity;

    transform.zeroMatrix();
    transform.rotateY(-1.57079f);
}
```

```
//TIME COUNT SINCE INPUT BEGAN
if (input.GetKey(ENG::KeyCode::W) || input.GetKey(ENG::KeyCode::S) || input.GetKey(ENG::KeyCode::A) || input.GetKey(ENG::KeyCode::D))
{
    timeSinceStart += t;
}
else if(timeSinceStart > 0.0f)
{
    timeSinceStart-= (t * 5.0f);
}

//UPDATE POSITION
if (input.keyWasPressed)
{
    velocity = (acceleration*timeSinceStart);

    if (velocity.x <= -maxVelocity)
        velocity.x = -maxVelocity;
    if (velocity.x >= maxVelocity)
        velocity.x = maxVelocity;

    if (velocity.z <= -maxVelocity)
        velocity.z = -maxVelocity;
    if (velocity.z >= maxVelocity)
        velocity.z = maxVelocity;

    setPosition(getPosition() + (velocity * t) + (0.5f * acceleration * (t * t)));

    std::cout << "Velocity X: " << velocity.x << ", " << " Velocity Z: " << velocity.z << "\n";
}
acceleration = glm::vec3(0.0f);
velocity = glm::vec3(0.0f);

//INSURES TIME NEVER GOES NEGATIVE
if (timeSinceStart <= 0.0f)
{
    timeSinceStart = 0.0f;
}
```

Morphing

Playing new and more advanced games, you notice certain aspects of the game that just appeal to the brain because of its natural way of existence. What this means is that the player will see things that just simply make sense the way they are supposed to. This includes things like running makes you move faster, and that wind blowing on an object will make it sway with the wind, like a flag. This is accomplished by morphing target to achieve the desired animation and appeal of it. How this works is a series of joints or morph target on an object that are manipulated in a set of keyframes and repeated through the cycle of those frames occurring. Running animations involve the character moving most of their body but primarily their legs and a cyclical fashion, and that cycle can be looped for the length that the character is running for. This applies to all animations in the sense that they can be replayed from the end point to the start point to create the effect of fluidity in an animation instead of it stopping and start again. How this applies to our game is the motion of our character from idle to walk to idle. This set of animations is where the morphine occurs, as well as in the walk cycle of the character. From idle, when the player begins to walk, they will move one foot in front of the other and continue with that cycle until they begin to idle again in which one foot will stop and the other will follow. Morphing each joint along the character allows us to perform these tasks in a smooth fashion, because when in walk, not just the legs move, but the joints in the hips and even up to the arms and shoulders. Now obviously this will vary with the type of character you are producing, for us it is more cartoon like so the arms are dragged behind the player instead of alternating like legs. The ideal effect of this is to have to character perform exactly what the player expects it to do when they press a button. The hard part comes in making fluid simple motions so that the player can truly understand what they are doing when they are doing it.



Speed Control

The ability to make object move is specific to the equation we are using to make it move in such a way. For example, our moving lights function off of curves and linear interpolation formulas, whereas our character does not because he is player controlled. Speed control affects the rate at which the object follows the path determined by said curve or interpolation. In term of physics and math, the objects are traveling a distance over a given period of time, meaning they have a speed at which they move. This means that we can control the speed of an object on the given path. One particular thing about speed control is it forces the object to travel from point to point at a constant speed the whole time. So once established, the object will travel at that speed the entire time, meaning if something is positioned farther away, it will have to speed up to account for the time it takes to get there. This makes is so that when an object reaches a point, it will not stall at the apex of its path, it will be forced to continue its desired path at a constant speed. This creates problems for puzzles in the game as well as the player playing. They are related in the sense that certain short paths the object would travel may be too fast for the player to truly get passed the obstacle. This method helps when have the most direct path for an object at the exact speed that we require it to be.