CMP502-Programming For Games

**Task 1: Build a Graphics Application**

**Using the following DirectX functionality**

1. Generating and rendering custom vertex geometry
2. Loading and rendering meshes
3. World, view and projection transformations
4. Lighting
5. Textures and materials
6. Camera control using keyboard
7. **Two** of the following features:
   1. Shader –generated particle system
   2. Sound effects
   3. Lighting that is more advanced than Phong-Blinn
   4. Smooth rotation using quaternion interpolation
   5. Loading and running skeleton animation from mesh file

Any other similar extra item you are interested in, discuss with me.

### Task 2: Documentation

### You must provide documentation for your application. This documentation should include:

1. Summary of the application ( Single paragraph)
2. Information on user controls
3. Technical insight into what you achieved and the techniques you used to achieve them. This is where you describe what you did and HOW you did it. There is no need to re-explain the techniques you have been taught in the lecture but I want to know what you did and how you went about it. If you have used a technique that was NOT covered in the material I expect you to be able to explain it and its implementation to demonstrate your understanding of what you have done.
4. Provide information on any shortfalls of your application. For example if normals are calculated incorrectly, explain why this is the case and try to suggest solutions to the problem. This is in effect a reflection on your work. What went right what went wrong and why?
5. References for any code or techniques incorporated in your application. Cases of plagiarism will be taken very seriously.

Things I do not need in report: Huge chunks of pasted code, there is no need. You can refer to functions classes etc but don’t paste the code into your report. Report is expected to be around 6 pages. Not counting cover, contents pages etc.

Summary:

This is a directx project making the use of the DirectXTK (Direct X Toolkit) to develop a graphical windows application.  
A space scene where the player can fly around in a jetpack to explore the environment.

The scene contains the 8 planets in the solar system which are not to scale but the distances and sizes are proportional. Rotations and revolutions have accurate angles.

There’s a rocket going around the sun with a particle system as it’s booster.

This report presents a DirectX application developed using the DirectXTK (Direct X Toolkit) to simulate a space environment within a graphical Windows application. Users can navigate the solar system with a jetpack, exploring proportional representations of the eight planets, which, while not to scale, maintain accurate relative distances and sizes. The planets’ rotations and revolutions are animated with precise angles to reflect true astronomical movements.

An interactive element is a rocket orbiting the Sun, enhanced by a particle system to mimic its booster’s exhaust. This feature demonstrates the application’s dynamic capabilities and DirectX’s power in rendering particle effects, adding depth to the user’s space exploration experience.

User Controls:

Q – Move Down

E – Move Up

W – Move Forward

S – Move Down

A – Move Left

D – Move Right

Left shift – Boost

Features:

- Loading and Rendering .obj meshes

One of the features I tried to implement was loading objects with multiple sub-meshes as I wanted to load models that I’d created.

This is one of the problems I faced when I tried loading in sub-meshes from .obj files by reading in the ‘o’ parameter and processing individual objects and adding them to an index and vertex buffer array so they can be drawn individually. Implementation of the arrays caused buffer errors which I couldn't debug further after trying for a bit with no luck due to time limitations.

The DirectX application includes a feature for loading and rendering .obj mesh files, which are commonly used to represent 3D models. The objective was to enable the application to handle objects composed of multiple sub-meshes, allowing for the integration of complex models I’d created.

During the implementation, a significant challenge arose when attempting to load sub-meshes from .obj files. The process involved reading the ‘o’ parameter within the .obj file format, which denotes individual object names. Each object was then processed to populate separate index and vertex buffer arrays. These arrays are essential for drawing each sub-mesh independently within the application, providing the necessary granularity for rendering detailed models.

However, the implementation of these arrays led to buffer errors that proved difficult to resolve. Despite efforts to debug the issue, the errors persisted, and due to time constraints, further investigation was not feasible. This limitation highlights the complexities of working with .obj files.

- Applying Textures with alpha

Another thing I tried to implement was transparent objects from textures by mapping the alpha channel. (Brayanzarsoft reference). In the pixel shader, I created a condition where if the texture on the current pixel is lesser than 1, it calculates the average colour and multiplies it to the light and ambient colour. On the other hand, to get the transparency, the alpha channel is added to the average colour and mapped to the alpha of the colour. This can give perfectly transparent materials, glass-like materials and even transparency with colour.

The application’s rendering capabilities were enhanced by implementing alpha transparency in textures. This feature allows for the creation of materials with varying degrees of transparency, from fully transparent to translucent.

The implementation involved the use of the alpha channel in textures as a means to determine transparency. In the pixel shader, a conditional check was introduced to evaluate the alpha value of the current pixel’s texture. If the alpha value was less than 1, indicating some level of transparency, the shader calculated the average colour of the pixel. This colour was then combined with the scene’s lighting and ambient colour. For pixels where transparency was required, the alpha channel’s value was added to the average colour calculation. This value is then mapped to the alpha component of the final pixel colour, allowing for precise control over the material’s transparency. The result is a versatile rendering technique that can produce a range of transparent materials, from completely clear to coloured translucency.

The development of this feature was guided by references such as Brayanzarsoft (1), which provided valuable insights into the intricacies of shader programming and texture mapping. Despite the complexity of the task, the successful implementation of alpha transparency significantly contributes to the realism and visual appeal of the rendered scene.

- Player Controls using Keyboard and Mouse

I started by using the screen dimensions to convert the mouse coordinates which are between -1 and 1 on both axes to fit between 0 and 1 to pass onto the camera to use as rotation values. This worked, but not as good as I wanted it to be. So, I used the ratios to calculate the rotation which I clamped the y axis between 90 and -90. This worked well for vertical look rotation. After further testing I figured out its limitation where even if the mouse was set to invisible, the user couldn’t rotate further horizontally if the mouse hit the edge of the screen. I now knew the problem and after another iteration, I figured out rotation by setting the mouse mode to relative so that I could get the slightest movements without having to worry about it to hit the edges of the screen. I then multiply the movements with the camera rotation speed, clamp the rotation in y axis and set the rotation vector. Using this value, I convert it to radians to calculate the forward direction of the camera.

Implementing intuitive player controls was a key feature of the DirectX application, allowing users to navigate the space environment using keyboard and mouse inputs. The initial approach involved converting the mouse coordinates on screen, which range from -1 to 1 on both axes, to a 0 to 1 scale using the dimensions of the window. These values were then passed to the camera as rotation inputs. While functional, this method did not provide the desired level of control precision. To improve the controls, the y-axis rotation was clamped between 90 and -90 degrees to ensure a natural vertical look rotation. However, testing revealed a limitation: horizontal rotation was restricted if the cursor reached the screen’s edge even when the mouse cursor was invisible.

The breakthrough came with setting the mouse mode to relative, which allowed for the capture of even the slightest movements without the cursor hitting the screen’s edges. Movements were multiplied by the camera’s rotation speed, and the y-axis rotation was clamped to maintain a realistic range of motion. The resulting rotation vector was then converted to radians to accurately determine the camera’s forward direction. This final iteration of the control system provided a seamless and responsive user experience, enabling fluid navigation through the virtual space environment.

- Smooth Rotation and Movement

Now that I got rotation to work properly, it was time to implement smooth rotation. I created a function which takes in the new rotation value and performs a quaternion spherical linear interpolation between the current and new values before sending it over to calculate the forward direction. This worked perfectly giving me smooth rotation after which I calibrated the ‘t’ (interpolation value) variable to my liking to achieve the effect. The rotation was no longer jittery by moving to every slight touch to the mouse.

The movement felt jittery for a space scene. To fix that, I created a direction vector where all the inputs are added. They are further normalized to prevent a well-known bug where the player moves faster if both the forward and directional buttons are pressed due to the vectors being added up. After which it is multiplied with the move speed. This is then added to the position vector and sent to the function which linearly interpolates between the old position and the new one this giving us smooth movement. Reducing the ‘t’ (interpolation value) variable to 0.01 gives a very free-floating movement. Much like the floating movement we have in space, and this fit the theme.

Achieving smooth rotation in the DirectX application was a critical step in enhancing the user experience. The solution involved creating a function that takes the new rotation value and applies quaternion spherical linear interpolation (SLERP) between the current and new rotation values. This method ensures a gradual transition, eliminating the jittery effect caused by minor mouse movements. The function performs the interpolation by calculating the shortest path on the unit sphere between the two quaternions representing the current and new rotations. The interpolation factor ‘t’ is then calibrated to fine-tune the smoothness of the rotation, resulting in a fluid and responsive camera movement that aligns with the user’s expectations for a space simulation.

To address the jittery movement, a direction vector was introduced to accumulate all input commands. This vector is normalized to avoid the common issue of increased movement speed when multiple directional inputs are active simultaneously. The normalized direction vector is then scaled by the movement speed and added to the position vector. The updated position is processed through a linear interpolation (LERP) function that blends the old and new positions, providing a smooth transition. By setting the interpolation value ‘t’ to 0.01, the movement achieves a free-floating sensation akin to zero-gravity conditions in space. This adjustment not only improves the realism of the application but also reinforces the thematic consistency of the space environment.

- Particle System

Rastertek reference

A particle system is an effect used for various effects that uses small particles such as rain, fire, smoke etc. Usually, an image(s) with transparent background is placed on a plane and duplicated several times to simulate small particles at a low cost. They are rendered as a billboard so that it always faces towards the camera in order to not break the effect when viewed from other angles.

I started by creating a “Particle system” class defining the properties of a particle which included the position, colour, velocity and whether it was active or not. This is done to save memory as particles are usually generated in large quantities (usually hundreds). Destroying and re-creating them every frame is computationally expensive. Thus, when a particle is created, it is added to a pool of particles. After it finishes its lifetime, it is disabled. When a new particle is needed, it goes through the list of particles and selects the disabled ones to render again with new values, which saves a lot of memory.

Since computations done in the shader is gpu dependent, it relieves the cpu of a lot of processing while rendering which would otherwise cause the cpu to calculate hundreds or thousands of positions and rotations. To pass the information to the shader, I create a “VertexType” struct which holds the same information for each particle. The class has an initialize function which will be called from “game.cpp” and does 2 things. Firstly, initialize the particles with different variables that would affect it during its lifetime such as particle deviation, velocity, variation, size, particles per second and max particles. An array is created to store these particles and they’re set to active. Finally, the function creates the vertex and index buffer which will be sent to the shaders. The render function takes the buffers and sets it to the shader.

The frame function runs every frame and performs 4 functions, kill, emit update particles and update buffers. The kill particles function checks if the particle is active and its local ‘Z’ position or accumulated time and decides if the particle must be disabled so that it can be reused later by the emit particles function. This function takes in the frame time and adds it to the accumulated time. It then checks if the time is greater than the number of particles per second. If a particle needs to be emitted, it sets a random position, velocity and the colour. It looks through the particles array and finds one which is inactive to enable or else creates a new particle. Since the particles are transparent and the order of rendering matters for if a particle which is ahead of another one is rendered before the particle behind it, it’ll cause the particle behind to be clipped due to the quad in front occluding it. So, before enabling the particle, I sort the particles array and insert the particle which has to be emitted based on the z-depth after which it is set to active with all the properties mentioned above. I wanted the particles to look like an exhaust from a rocket, so in the update function I add velocity to the local position. The final step is to update the buffers by mapping the vertex data to be passed to the shaders and rendered.

To render the particles, I created a vertex shader which takes in the position, texture coordinates and the colour to calculate the position against the world, view and projection matrix. It further passes on the position, texture coordinates and colour to the pixel shader which samples the texture with alpha values and sends the final colour for the current pixel.

We need a particle shader class to pass the information from the buffers to the shader. The initialize function is called from the “game.cpp” file. It takes in the compiled vertex and pixel shader files to create the respective buffers. The data being passed to the buffers must match the structures that were created in the respective shaders. This class also contains a render function which is called every frame and takes in the world, view and projection matrices along with the texture to be applied. This function performs 2 functions. Firstly, setting the shader parameters by adding the matrices to the matrix buffer and set the texture to the shader resources slot. Finally, it calls the render shader function which sets the vertex shader, pixel shader, sample state and draw the vertices on screen.

To make the effect look better, by making it brighter if there are multiple particles like a rocket exhaust in real life. I set the blend state additive before calling any of the render functions. This enhances the effect and it looks much better. A lot can be done with particle systems by manipulating geometry, drawing order and playing with the emit settings. I tried to add a billboard system so that it always faces the camera using the cameras forward direction. But due to the particle system always in motion, it was harder than I expected. I also wanted to make a dynamic sorting system for when the particles are moving in the z direction as currently, they are sorted only when they are spawned.

I implemented a particle system in the DirectX application guided by a tutorial by Rastertek (2), an effect commonly used to simulate phenomena such as rain, fire, and smoke. This system relies on small particles, represented by images with transparent backgrounds placed on planes, which are duplicated multiple times to create the illusion of numerous particles. To ensure the effect remains consistent from various angles, these particles are rendered as billboards, always facing the camera.

I began by creating a “Particle System” class to define the properties of a particle, including its position, colour, velocity, and active status. This design choice was made to conserve memory, as particles are typically generated in large numbers. Rather than destroying and recreating particles each frame, I added them to a pool. When a particle’s lifetime expired, it was merely disabled, not destroyed, allowing for its reuse by reactivating it with new values when needed.

The computations for the particle system are handled by the GPU within the shader, which significantly reduces the CPU’s workload. I passed information about each particle through a “VertexType” struct to the shader. The class’s initialization function, which I called from “game.cpp”, which performs 2 tasks. Firstly, initialize the particles with different variables that would affect it during its lifetime such as particle deviation, velocity, variation, size, particles per second and max particles. An array is created to store these particles and they’re set to active. Finally, the function creates the vertex and index buffer for communication with the shaders. The render function takes the buffers and sets it to the shader.

Each frame, the frame function performs four essential tasks: killing, emitting, updating particles, and updating buffers. The kill function deactivates particles based on their local ‘Z’ position or accumulated time, making them available for re-emission. The emit function, takes in the frame time as a parameter and adds it to the accumulated time. It then checks if the time is greater than the number of particles per second. If a particle needs to be emitted, it assigns randomized positions, velocities, and colours to particles. It selects inactive particles from the array for reactivation or creates new ones as necessary. Since the particles are transparent and the order of rendering matters for, if a particle which is ahead of another one is rendered before the particle behind it, it’ll cause the particle behind to be clipped due to the quad in front occluding it. So, before enabling the particle, I sort the particles array and insert the particle to be emitted based on the z-position after which it is set to active with all the randomized properties mentioned above. In the update function, I add velocity to the local position to simulate rocket exhaust, and the final step involves updating the buffers with vertex data for rendering.

To render the particles, I developed a vertex shader that processes the position, texture coordinates, and colour of each particle. This shader calculates the particle’s position against the world, view, and projection matrices, ensuring correct placement within the 3D space. The data is then passed to the pixel shader, which samples the texture with alpha values to determine the final colour of each pixel.

I used a particle shader class to transfer particle data from the buffers to the shaders. This class, initialized from the “game.cpp” file, creates the necessary buffers based on the compiled vertex and pixel shader files. The data structure passed to these buffers must match the structures defined in the shaders. The class’s render function, called every frame, sets the shader parameters by updating the matrix buffer with the world, view, and projection matrices and assigning the texture to the shader resource slot. It then executes the render shader function, which activates the vertex and pixel shaders, sets the sample state, and draws the vertices on the screen.

To enhance the visual impact of the particle system, I set an additive blend state before any rendering. This blending mode increases the brightness of overlapping particles, creating a more intense and realistic effect. Although I faced challenges with the billboard system and dynamic sorting, the particle system’s implementation demonstrates the application’s ability to manipulate geometry and control drawing order to create compelling visual effects. I aim to achieve a billboard system that consistently faces the camera and a dynamic sorting mechanism for moving particles in future developments.

- Sound Effects

I used XAudio 2.9 in the directx toolkit for audio by adding in the defines in “pch.h” and defining “DXTK\_AUDIO” in the “game.h” file. After creating and initialising the audio engine class when an audio device is found, we can use the classes which derive for it which include a wave bank and a sound effect class. Wave banks are an efficient way of managing audio files by packaging them into a “wave bank” which allows for more efficient loading and memory optimisation. The audio engine’s update function is called every frame so that it is able to play any audio at any time ranging from background music, looping tracks to one shot audio effects. I wanted the scene to feel more responsive to the players actions by adding in audio for the jetpacks when they more around. I load in 2 audio files from the wave bank for normal jetpack audio and when the player uses the “left shift” key to boost. I created a variable “isMoving” to check when the player pressed down a key and play the jetpack sound effect. If the player is moving, I also check when the player boosts to play the boost audio on top of it. For the rocket ship going around the sun, I implemented 3d audio by first loading the rocket sound effect from the wave bank and created a sound effect instance with the flag “SoundEffectInstance\_Use3D”. After creating the variables audio emitter and the audio listener, in the update function, I update the listeners position and rotation to the camera’s position and rotation. I then get the position of the rocket by decomposing the matrix and set the position of the emitter to it. After playing the audio, I then apply 3d audio by passing in the listener and emitter variables. On playing, we can hear the rocket go around and if we turn the camera we can hear the audio coming from its direction.

In my DirectX application, I integrated sound effects to enhance the user experience using XAudio 2.9 from the DirectX Toolkit. By defining “DXTK\_AUDIO” in the “game.h” file and setting up the necessary preprocessor directives in “pch.h”, I was able to initialize the audio engine once an audio device was detected. The engine utilizes wave banks and sound effect classes derived from it, which are efficient in managing audio files. Wave banks package audio into a single file for optimized loading and memory usage.

The audio engine’s update function is called every frame, enabling the playback of various audio types, from background music to one-shot effects. To make the scene more interactive, I added sound effects for the jetpacks that respond to player movement. Two audio files are loaded from the wave bank: one for the standard jetpack sound and another for the boost effect triggered by the “left shift” key. I implemented a variable “isMoving” to detect when the player is in motion and play the corresponding jetpack sound effect. Additionally, when the player boosts, the boost audio overlays the standard jetpack sound.

For the rocket orbiting the sun, I introduced 3D audio to create a more immersive environment. I loaded the rocket sound effect from the wave bank and created a sound effect instance with the “SoundEffectInstance\_Use3D” flag. I then established audio emitter and listener variables, updating the listener’s position and rotation to match the camera’s in the update function. By decomposing the rocket’s matrix, I obtained its position and set the emitter’s position accordingly. Upon playing the audio, I applied 3D audio effects using the listener and emitter variables, allowing players to perceive the rocket’s movement around the sun audibly, with the sound direction changing as the camera turns.

This implementation of sound effects significantly contributes to the realism and responsiveness of the application, providing an engaging auditory experience that complements the visual elements.

- Creating the solar system

Since all planets are spheres, I use a unit sphere and scale it to a size proportional to the other planets. I needed to get the planets to rotate on their own axis, revolve around the sun on their own axis’. To achieve this I had to multiply the matrices in the right order. I use the following order to get most of the planets in place. Starting with the scale, I multiply it with the local rotation speed on its y axis followed by the local axial tilt. This is multiplied with the world matrix followed by the distance from the sun, revolution speed and then the revolution axis. I created a switch case for all the planets and introduced a variable “submodel” For planets like the earth, where I needed to add a moon and use already calculated positions of the earth and post-multiply it with the variables of the moon. I achieved this by using this equation as the world matrix: (moon) (scale \* rotation \* axial tilt \* position \* revolution \* revolution axis) \* (earth) (position \* revolution \* revolution axis). I used a similar approach for Saturn's rings. I had to create the asteroid belt between Mars and Jupiter which is like a particle system but using 3d objects. It had to look random enough with different shapes and sizes of rocks along with local rotations. I created an asteroid class with some initialization variables such as the model, scale, distance, local rotation, axis of rotation, rotation speed and revolution speed. In the render function of the game class, I check if the asteroids have been initialized. If they haven’t, I set the number of asteroids to spawn and create an asteroid array with the size of the number of asteroids. I then loop through the array and initialize all the variables with random values. Starting with the model, I choose from 6 different models and assign it. The scale is a random value between 0.08 and 0.75. The local rotation is a vector 3, so I created a for loop to assign random local rotation values. This is followed by randomizing the axial speed which is a vector 2 where I set the ‘x’ value to be an integer for the rotation axis and the ‘y’ value as a float for the rotation speed. The asteroids distance variable is then initialized so that it is placed between Mars and Jupiter but closer to Mars. Finally, I set a Random revolution speed and set the initialized variable to true after all exiting the loop. Now I can loop through all the asteroids and set the initialized values. There’s a switch case to choose the local rotation axis. The order of multiplication for the asteroid is similar to the planets above. I set the model variable as the asteroid and render it with a texture.

I researched about the sizes, distances, angles for axis of rotation, angles for revolution, rotation speeds for how long days are and revolution speeds to provide a fairly accurate representation of the solar system.

In creating the solar system for my DirectX application, I began with a unit sphere to represent the planets, scaling each to a size proportional to the others. To simulate the planets’ rotation and revolution, I carefully multiplied transformation matrices in a specific order. The process started with scaling, followed by multiplying with the local rotation speed on the y-axis and the local axial tilt. This combination was then multiplied with the world matrix, the distance from the sun, the revolution speed, and finally, the revolution axis.

For planets with additional elements, like Earth and its moon, I introduced a “submodel” variable. I used the already calculated Earth’s position as a base and post-multiplied it with the moon’s variables using this equation as the world matrix: (moon) (scale \* rotation \* axial tilt \* position \* revolution \* revolution axis) \* (earth) (position \* revolution \* revolution axis). A similar approach was applied to Saturn’s rings.

The asteroid belt posed a unique challenge, as it required a particle system-like approach but with 3D objects to simulate randomness in shape and size. I created an asteroid class with some initialization variables such as the model, scale, distance, local rotation, axis of rotation, rotation speed and revolution speed. In the render function of the game class, I check if the asteroids have been initialized. If they haven’t, I set the number of asteroids to spawn and create an asteroid array with the size of the number of asteroids. I then loop through the array and initialize all the variables with random values. Starting with the model, I choose from 6 different models and assign it. The scale is a random value between 0.08 and 0.75. The local rotation is a vector 3, so I created a for loop to assign random local rotation values. This is followed by randomizing the axial speed which is a vector 2 where I set the ‘x’ value to be an integer for the rotation axis and the ‘y’ value as a float for the rotation speed. The asteroids distance variable is then initialized so that it is placed between Mars and Jupiter but closer to Mars. Finally, I set a Random revolution speed and marked the initialization as complete.

Looping through the asteroids, I applied the initialized values and used a switch case to determine the local rotation axis. The multiplication order for rendering the asteroids was similar to that of the planets. I set the model variable to the asteroid and rendered it with a texture. My research into the solar system’s sizes, distances, rotational and revolutionary angles, and speeds ensured a fairly accurate representation within the application.

- Skybox

I set the background to black but it wasn’t enough to fit the space vibe. So, I decided to implement a simple Skybox which had transparent textures and a parallax effect. I created a sphere in blender with the normals facing inwards and exported it to the project. Creating 2 of them with the size of the far plane and applying different rotations for them make the effect much better. I added the camera translation to the Skybox so that the effect doesn’t break and achieved a very simple Skybox.

To enhance the space ambiance of my DirectX application, I opted to implement a Skybox with transparent textures and a parallax effect. I started by creating a sphere in Blender and some textures in photoshop, ensuring the normals were facing inwards to simulate the vastness of space from an interior perspective. After exporting the sphere into my project and applying the texture, I duplicated it to create two layers. These layers were scaled to match the size of the far plane, and I applied distinct rotations to each to amplify the parallax effect.

To maintain the illusion of a boundless universe, I incorporated the camera’s translation into the Skybox’s transformation. This technique ensures that the Skybox moves with the camera, preventing any visual discontinuities that could disrupt the immersive experience. The result was a simple yet effective Skybox that significantly contributed to the overall space-themed environment of the application.