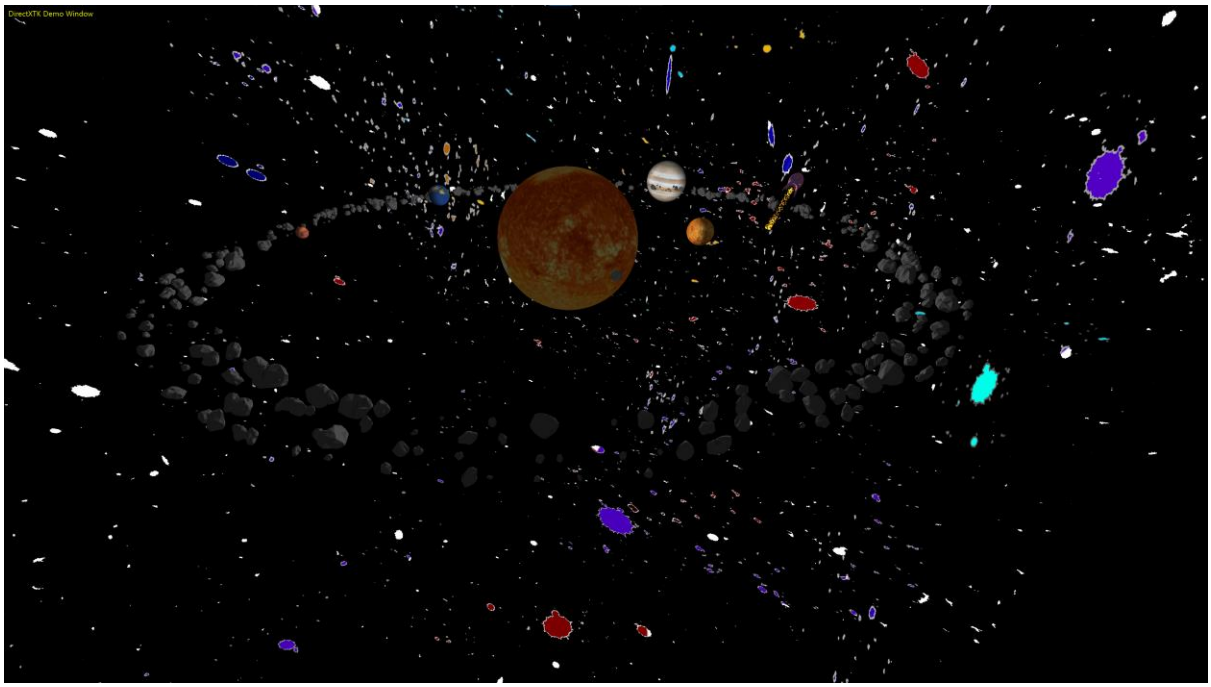
DirectX Solar System Project Documentation

Shubh Ravishankar Gawhade

MSc Computer Games Technology, 2023-24



DirectX Application Screenshot

School of Design and Informatics

Abertay University

Contents

Summary 3
User Controls 3
Features 4
- Loading and Rendering .obj meshes 4
- Applying Textures with alpha 4
- Player Controls using Keyboard and Mouse 5
- Smooth Rotation and Movement 5
- Particle System 6
- Creating the solar system 7
- Sound Effects 8
- Skybox 9

References 10

Summary

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

P – Pause

Esc – Quit

Features

Loading and Rendering .obj meshes

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

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

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 mouse coordinates on the 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 "MODE_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

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

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 in 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”, 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 them to the shader.

For 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, 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.

Creating 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 by 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 asteroid's 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.

Sound Effects

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.

Skybox

To enhance the space ambience 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.

References

[1] Braynzar Soft (no date) Braynzar Soft - game programming tutorials and questions! Available at: https://www.braynzarsoft.net/ [Accessed: 25 November 2023].

[2] Rastertek (no date) RasterTek - DirectX 11 and OpenGL 4.0 tutorials. Available at: https://rastertek.com/ [Accessed: 2 December 2023].

[3] Walburn, C. (2022) DirectXTX Wiki, GitHub. Available at:

https://github.com/microsoft/DirectXTK/wiki/ [Accessed: 9 December 2023].