186.140 Echtzeitgraphik

King's Row

Rebeka Koszticsak, 1325492
J. Sebastian Kirchner, 0926076

Implementierung

Die Basis vom Code ist das CGUE SS15 Projekt von Rebeka Koszticsak und Peter Pollak.

Rendering: Mit dem Rendering zusammenhängende OpenGL Funktionen sind in der Renderer Klasse gesammelt. Bei der Initialisierung der Klasse wird das benutzte OpenGL Profile (3.3) festgelegt. Objekte sind in der MeshNode Klasse gekapselt und enthalten sowohl alle Objektdaten (Vertices, Texturen, …), sowie Buffer Informationen für die Daten auf der Grafikkarte. Durch eine prepare Funktion werden diese Daten der Renderpipeline übergeben und auf die Grafikkarte geladen. Zusätzlich werden diese Objekte in einem Array gesammelt, die in der Rendering Schleife durchiteriert wird. In der Schleife wird die draw Funktion jedes Objektes aufgerufen, sowie die view und projection matrix aktualisiert. Daraufhin übergibt sich das Objekt, zusammen mit Informationen über die Position, selbst, an die draw Funktion der Renderer Klasse, welche den richtigen Shader und Buffer aktiviert, zeichnet, und wieder deaktiviert. Sobald alle Objekte gezeichnet wurden, werden diese durch glfwSwapBuffers auf den Bildschirm geladen.

Beleuchtung: Die Shading ist sehr modular implementiert. Wenn die Objekte geladen werden, wird auch ihr Shader festgelegt. Es ist auch möglich, dass unterschiedliche Objekte, unterschiedlich geshadet werden. In unserem Code wird derzeit für jedes Objekt ein Blinn-Phong Shader verwendet, modifiziert durch den Effektshader, der Directional-, Spot-, und Pointlights unterstützt. Die MeshNode Klasse speichert das benutzte Shader-Program (Klasse ShaderProgram), welches erzeugt wird, wenn das erste Objekt geladen wird, und für weitere Objekte wiederverwendet wird. Während dem Rendern wird immer der Shader des aktuellen Objektes aktiv gestellt. Der Renderer lässt das Shader Program die benötigte Uniforms befüllen, wozu Daten aus dem Objekt, sowie die aktiven Lichter der Szene benötigt werden.

Texturen: Ähnlich zum Shader, wird die Textur des Objektes beim Object Loading festgelegt und in der MeshNode Klasse mitgespeichert. Zur Initialisierung eines Objektes wird der Pfad der Textur benötigt. Dadurch kann die Klasse Texture initialisiert werden, deren Aufgaben ist die Textur zu laden. Wenn die Textur während dem shading benötigt wird, kann die ShaderProgram Klasse auf diese Informationen durch die MeshNode Klasse zugreifen.

Camera: Die Klasse CameraNode steuert grundsätzliche Kameraeigenschaften wie die View- und Projection-Matrix. Kameras werden in der Main Klasse in einer map gespeichert und können aktiviert werden, in dem sie als die activeCamera Pointervariable gesetzt werden. Die aktuelle Szene enthält jedoch nur eine Kamera, welche an derselben

TransformNode wie die PlayerNode hängt. Die Kamera selbst ist statisch und kann ihre Position nicht ändern, die PlayerNode welche sich die TransformNode mit der Kamera teilt, kann diese jedoch verändern. Vor jedem draw call wird der Input abgefragt und beim updaten der Szene an jede Node des Szenegraphens übergeben, die PlayerNode ändert ihre TransformNode abhängig vom Input, wodurch auch die Kamera Position und Orientation geändert wird.

Szene: Die Objekte werden im Szenengraphen(SceneNode) verwaltet. Jedes Objekt (Mesh, Kamera, Licht) muss im Szenengraph enthalten sein. Jeder Objekttyp hat eine entsprechende Klasse welche beim Laden der Szene bekannt sein muss, von SceneNode erbt, in den Szenengraphen integriert werden kann und typenspezifische Funktionen Implementiert. Zum Beispiel verwaltet die CameraNode die View- und Projection-Matrix, die PlayerNode reagiert auf Input, die MeshNode enthält alle nötigen Informationen zum Rendern wie die Speicherposition auf der Grafikkarte, Shader, Texturen usw. Die Position der Objekte in der Szene werden durch TransformNodes bestimmt an welche die Objekte angehängt werden. Zu der vollständigen Initialisierung müssen alle Knoten verknüpft werden, wobei Ringe im Graphen nicht zulässig sind und der einzige Knoten, keinen Parent hat, die RootNode ist von welcher die komplette Szene ausgeht.

Effekte:

Light Shafts: Die Light Shafts werden durch die Klasse LightShaft.h gesteuert. Diese Klasse speichert die Framebuffers und hat die Methoden sie im richtigen Zeitpunkt zu akrivieren. Für Rendering von Light Shafts werden 3 Rendering Passes benötigt. Erstmal wird die Szene normal, mit dem üblichen Shader, in den ersten Framebuffer gerendert. Als zweites wird die Lichtquelle von den Light Shafts und die Objekte gerendert, die diese Lichtquelle verdecken. Diese Informationen werden in den zweiten Framebuffer geschrieben. Zum Schluss stellt die Klasse LightShaft beide Framebuffers zur Verfügung und die beide Szenen werden kombiniert. Es wird überprüft, ob vom derzeitigen Pixel die Lichtquelle sichtbar ist, und falls ja, wird die Helligkeit abhängig vom Abstand interpoliert, und die entstandene Strahlen werden zu dem originalbild addiert.
Die Grundidee von der Light Shaft Implementierung und der Aufbau von den Shaders stammt von: http://zompi.pl/light-shafts-b/.

Wasser:
Disclaimer: Der Wassereffekt konnte leider noch nicht fertig gestellt werden, die ersten beiden Renderpasses rendern eine schwarze Textur und der Fehler konnte leider noch nicht behoben werden.

Um das Wasser zu rendern werden zwei zusätzliche Renderpässe benötigt. Einmal wird alles in eine Textur gerendert, was oberhalb des Wassers ist (reflection), und einmal alles, das unterhalb des Wassers ist (refraction). Die Klasse Water.h speichert die Framebuffer und Texturen für die reflection und refraction Render Passes. Im Fragment (WaterFragmentShader.glsl) Shader werden die beiden Texturen dann vermischt. Um einen korrekten Effekt zu erzeugen, muss projektives Texturen-Mapping angewendet werden.
Ein Welleneffekt sollte im Zuge einer DU/DV Map generiert werden (dieser ist allerdings leider noch nicht implementiert).

Die Grundidee von der Water Implementierung und der Aufbau von den Shadern stammt von: https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter19.html

Zusätzliche Libs

- Image Loading: std_image (https://github.com/nothings/stb/blob/master/stb_image.h)
- Object Loading: assimp (http://assimp.sourceforge.net/)
- Glad (http://glad.dav1d.de/)
- GLFW (http://www.glfw.org/)
- GLM (https://glm.g-truc.net/0.9.8/index.html)
- https://www.opengl.org/

Grafikkarte

Der Code ist auf NVIDIA implementiert.

Modelle

Die Modelle wurden soweit nicht anders gekennzeichnet mit Blender erschaffen und unwrapped.

Bäume: https://www.turbosquid.com/3d-models/free-tree-3d-model/592617
Brücke: https://free3d.com/3d-model/old-bridge-84920.html

Kamerabewegung

Das Framework liest einen File ein, wo die vordefinierten Input Werten für Kamerasteuerung gespeichert sind. Wenn das Dokument bis zum Ende gelesen wurde und die Kamera alle vordefinierte Werte durchgegangen ist, terminiert die Applikation automatisch. Das Dokument emuliert die Tastatur und Mausinput, die Tastatur und Maus wird während der Laufzeit nicht gelesen. Um die Werte im Dokument richtig zu verarbeiten wurde ein string spliting Alorithmus von http://www.cplusplus.com/articles/2wA0RXSz/ verwendet.