

# Dokumentation zur Abgabe 2

Pathracer 4096 - Gruppe 22

- Dennis Metzger (11809610)
- Andreas Melcher (00004876)

## Features

Man steuert ein Schiff im Weltraum an einem Pfad entlang. Der Pfad ist dabei von einem holografischen Schlauch umgeben. Per Tastatur oder Controller kann das Schiff in alle Richtungen (vor, zurück, links, rechts, rauf, runter) gesteuert und um die Längsachse gerollt werden.

Die Geschwindigkeit des Schiffes hängt dabei von der Entfernung zum Pfad ab. Fliegt man in der Mitte des Schlauches, fliegt man am schnellsten. Einige enge Kurven und ein paar Asteroiden die den Schlauch an machen Stellen kreuzen machen es allerdings nicht einfach längere Zeit am idealen Weg zu bleiben.

Ziel ist in einer möglichst kurzen Zeit eine Runde zu vollenden. Die momentane Rundenzeit und die bisherige Bestzeit werden rechts unten angezeigt und man kann versuchen seine Bestzeiten zu unterbieten.

Damit man im dunklen Weltraum keine Abkürzungen nehmen kann, befinden sich am Pfad entlang "Targetpoints" an denen man, in oder recht nahe am Schlauch, vorbeifliegen muss. Lässt man so einen Punkt aus, stoppt am Ende der Runde die Zeit nicht und eine Bestzeit ist kaum möglich. Nur wenn man alle Targetpoints in der richtigen Reihenfolge durchfliegt, wird die Zeit am Ende der Runde auch gestoppt. Erkennen kann man die Targetpoints an einer feinen Wand aus weißen Lichtern.

Stößt man am Weg durch den Schlauch mit einem der Asteroiden zusammen, so läuft die Zeit einfach weiter, man wird aber zum vorherigen Targetpoint zurückgesetzt.

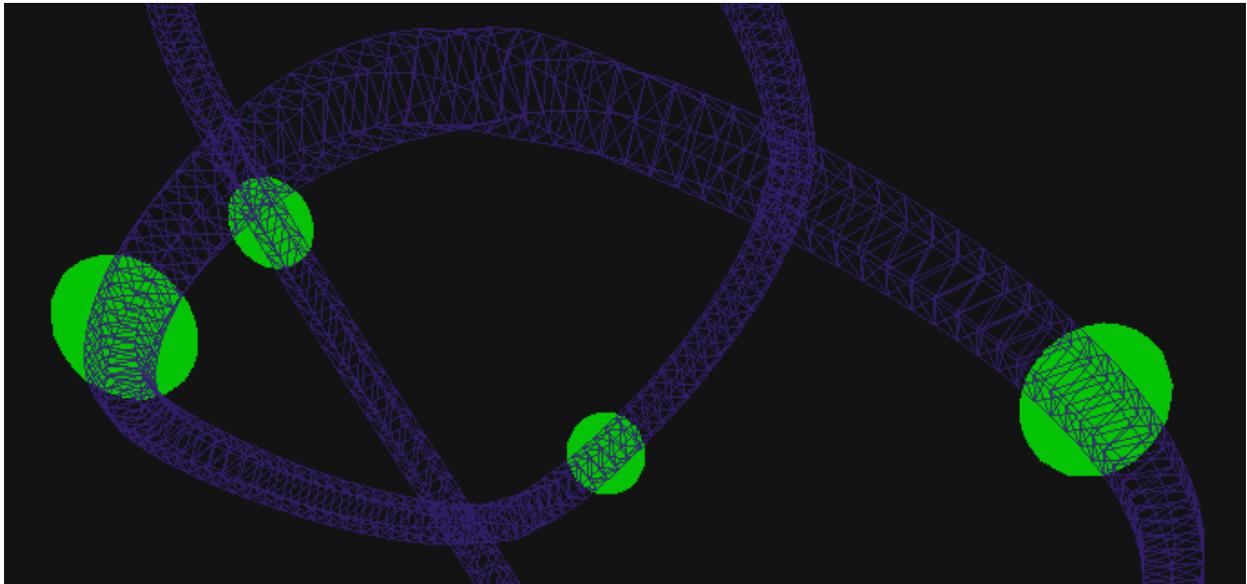
# Implementierung

## Der Pfad

Wir setzen bestimmte *PathControlPoints*, eine *StepSize*, *PathEdges* und einen *Radius*. Unsere *Path*-Klasse berechnet damit dynamisch die Pfadpunkte entlang eines [Catmull-Rom Splines](#). Mit *generateMesh* bauen wir einen *Triangle Strip* der eine Hülle um den Pfad im Abstand des mitgegebenen *Radius* berechnet. Je nach Anzahl der *PathEdges* wird ein Schlauch mit einer 3 bis n eckigen Schnittfläche erzeugt. Wir rendern diese Hülle mit Linien und transparenten Flächen.

## Collision Detection - Targetpoints und Asteroiden

Die abzufliegenden *TargetPoints* sollen erreicht sein, wenn man in einem bestimmten Abstand an ihnen vorbeifliegt. Daher sind die *Collider* dort kugelförmig gewählt. Unser Asteroiden Modell ist länglich weswegen hier ein *Collider* mit *Capsule Shape* verwendet wird. Das Schiff wiederum hat 3 *Collider* um die Form des Schiffs besser abzudecken. Der Rumpf ist ebenfalls eine *Capsule* die Flügel sind mit je einer *BoxShape* abgedeckt.



*Beispiel für die Collider der targetpoints entlang des Pfades*

Es gibt einen *CollisionListener* der von *EventListener* der *ReactPhysics3D Library* ableitet und der sich um die Folgen der Kollisionen kümmert. Berührt das Schiff den *Collider* eines solchen *Targetpoints* bzw einen Asteroiden, gibt es die Klasse *RacingInfoForOneShip* die den State des Schiffs im Rennen berechnet und beinhaltet. Wenn also zB ein *Targetpoint* durchflogen wird berechnet diese Klasse ob es der richtige war und ob zB eine Runde zuende ist. Auch die Zeitnehmung ist hier gekapselt. So ist eine einfache Erweiterung für mehrere Schiffe möglich, indem es für jedes Schiff eine eigene *RacingInfoForOneShip* Instanz gibt.

## Das Rendering

Wir verwenden unser eigenes Mesh Format (CgUe-Mesh). Unser MeshTool konvertiert jede Assimp ladbare Szene in x-Meshes und speichert diese in den angegebenen Ausgabeordner im Format "Meshname.CUM".

Als **Renderer** haben wir einen Mix aus **Forward und Deferred Rendering** implementiert, inklusive OIT Pass wodurch transparente Objekte ohne PBR einfach möglich sind zB der Pfad. Desweiteren verfügen wir über eine kleine Postprocessing Pipeline die Bloom & Screen-Space-Reflektionen umsetzt.

PBR-Shading wird beim Kombinieren der GBuffer (die Deferred-Buffer) umgesetzt.

Die World-Position jedes Pixels wird dabei rekonstruiert und entsprechende Lichtberechnung auf diesen ausgeführt.

Um nicht jeden Pixel beim Kombinieren zu zeichnen verwenden wir weiters einen Stencilbuffer, welcher markiert welche Pixel von unseren Objekten veranschlagt werden. Der Kombinations-Pass wird dann auch nur auf diese angewandt, dafür wird der Depth-Stencil-Buffer zwischen GBuffer und Backbuffer geteilt.

OIT Pixel werden ebenfalls im Kombinationspass gezeichnet. Standardmäßig ist der alpha Wert eines Pixels auf 0 gesetzt wodurch OIT Pixel diese überschreiben sollte kein opakes Objekt dorthin gerendert haben.

Das Physik-Debug-Rendering & Text-Rendering hingegen wird als klassisches Forward Rendering umgesetzt, was daran liegt, dass alle transparent gezeichnet werden und auf diese Weise einfach nach dem Kombinationspass gezeichnet werden können. Außerdem benötigen diese Darstellungen kein PBR.

Der Pfad verwendet OIT wie unten beschrieben.

# Umgesetzte Effekte

- PBR-Rendering (Schiff, Asteroiden)

Basierend auf: <https://github.com/Nadrin/PBR/blob/master/data/shaders/hlsl/pbr.hlsl>

Wir übergeben unseren PBR Objekten 4 Elemente an die jew. Shader.

Albedo, Emissive, Roughness, Metalness

Normal & Position entnehmen wir dem Vertexshader

Die ersten vier dienen für den Fall, dass es keine jew. Texturen gibt.

- Texturing

Texturen angewandt

- Teilw. Spec.Map

Wir zweckentfremden unsere Roughness Map/Werte vom PBR Durchgang um beim Screen Space Reflection Prozess die Reflektionen zu maskieren/abzuschwächen, sodass sehr roughe Oberflächen kaum/gar nicht reflektieren

- Linked Pixel Lists (Pfad, wir haben außerdem eine Testszene um es noch besser zu erkennen)

Unser Pfad wird komplett OIT gerendert. Dafür halten wir uns strikt an die AMD-Präsentation:

[http://developer.amd.com/wordpress/media/2013/06/2041\\_final.pdf](http://developer.amd.com/wordpress/media/2013/06/2041_final.pdf)

Wir erstellen einen shared-shader-buffer für die Linked Pixel

Dort speichern wir nur die endgültige Farbe des Pixels

Etwaige Schatten-/PBR-Berechnungen werden bereits im Pixelshader vor den opaken Objekten durchgeführt, wodurch auch nicht PBR Materialien in diesen OIT Buffer schreiben können wie eben bei unserem Pfad.

- Shadow Mapping (direkt am Schiff erkennbar, je näher der Sonne desto schärfer)

Wir haben dafür kein Tutorial oder dergleichen benutzt, sondern aus dem Bauch heraus geschrieben.

Normales Shadowmapping a lá für jedes Licht, dass Schatten wirft, die Szene in einen eigenen "Depthbuffer" rendern.

Jedes Licht bekommt eine eigene View-Projektionsmatrix, je nach dem welcher Typ (Spot-/Pointlight)

Directional Lights haben wir bewusst nicht implementiert.

Ein Unterschied zu gängigen Pointlight Implementierungen ist, dass wir auch dafür nur eine einzelne Textur (keine Cubemap) verwenden und anhand der Kameraposition berechnen "wohin das Licht schaut". Wir konvertieren so zu sagen ein Pointlight zu einem Spotlight

- Screen Space Reflections (schwierig auf die Distanz beim Schiff zu sehen, wir haben für das Abgabe-Gespräch eine Testszene dafür erstellt)

Basierend auf dem beiliegenden pdf "Screen Space Reflections"

Allerdings funktionierte die Referenzlösung nicht 1:1, weshalb wir den Effekt "normal" implementiert haben.

Wir gehen die Screen-Space Position entlang des Reflektionsvektors (ebenfall screen-space) Und überprüfen jeden Schritt ob eine Kollision mit dem Z-Konus entsteht.

Wir wollten ursprünglich komplett im Pixel-Space entlang gehen, aber da unser Weg bereits funktionierte blieben wir dabei. Wodurch allerdings Pixel mehrfach gesampled werden können :( Ein weiterer Unterschied zum pdf: wenn wir keine Kollision erhalten sampeln wir einfach die Skybox

- Bloom (leicht zu sehen bei der Sonne, den Modellen der Targetzones und den weißen Highlights des Schiffs)

Wir haben uns eine einfache Postprocess Pipeline gebaut

Jeder Postprocess bekommt als Input den GBuffer sowie den Backbuffer des vorherigen Effekts bzw des Renderings wenn 1. Effekt.

Beim Bloom haben wir nach: <https://learnopengl.com/Advanced-Lighting/Bloom> programmiert

Wir erstellen im Bloom-Process zuerst eine Brightness Textur die nur die hellsten Pixel enthält Diese Textur wird dann mehrfach vertikal & horizontal verwischt (= geblurt)

Am Ende wird die Blur-Textur mit der Input-Texture additiv kombiniert.

- Vertex Shader Animation (Sonne)

Ursprünglich war unser Ziel einen Wasser-Planeten aus einem einzelnen Ozean zu erstellen.

Bevor wir diesen implementieren wollten spielten wir allerdings mit Kosinus & Sinus Funktionen im Vertexshader und fanden, dass unser Ergebnis wie eine Sonne aussieht und blieben dabei.

Dafür haben wir im Vertexshader der Sonne (waterball\_vs) die Vertexposition entlang des Normalenvektor, multipliziert mit mehreren Cos-Sin-Frequenzen, verschoben. Außerdem verwenden wir die globale Zeit (gt) mit einer Pseudo-Zufallsfunktion und der Texturkoordinaten um etwas Abwechslung hinein zu bringen.

Im Pixelshader berechnen wir, mittels dFdX/dFdY-glsI Funktionen, die neuen Normalen.

Da die alten nur für eine glatte Oberfläche gültig waren.

## Steuerung des Schiffs

- WASD:
  - hoch, links, runter, rechts rotieren
- Leertaste:
  - vorwärts beschleunigen
- linke Ctrl/Strg
  - Bremsen / rückwärts fliegen
- Pfeiltasten Rechts/Links
  - um die Längsachse rollen

## Wie man durch das Spiel kommt

Einfach drauf losfliegen, keinen Targetpoint auslassen und dann versuchen immer bessere Zeiten zu fliegen. Wenn man es schafft im oder möglichst nahe am Schlauch zu bleiben hilft das dabei sehr.

## Special Features

Folgende Zusatzfeatures kann man mit Tasten togglen:

- F1: Debugging der Collider
- F2: Shadowmap anzeigen

## Eingesetzte Libraries

- ReactPhysics3D
  - <https://www.reactphysics3d.com/usermanual.html>
  - Collision Detection
- FreeImage
  - <https://freeimage.sourceforge.io/>
  - Texturen laden
- Glew
  - <http://glew.sourceforge.net/>
  - OpenGL funktionen laden
- GLFW
  - <https://www.glfw.org>
  - Context- & Fenster-Erstellung
- LibVorbisfile
  - <https://xiph.org/vorbis/doc/vorbisfile/index.html>
  - zum Laden von .ogg Audio-Dateien (nicht implementiert, aber exe linked dagegen)
- Libogg
  - <https://xiph.org/ogg/doc/libogg/index.html>
  - wird von libvorbisfile verwendet (nicht implementiert, aber exe linked dagegen)
- Libvorbis
  - <https://xiph.org/vorbis/doc/>
  - wird von libvorbisfile verwendet (nicht implementiert, aber exe linked dagegen)
- Assimp
  - <https://www.assimp.org/>
  - wird von unserem MeshTool zur Mesh-Bearbeitung verwendet
- ImGui
  - <https://github.com/ocornut/imgui>
  - MainMenu - WIP (nicht implementiert, aber exe linked dagegen)
- XAudio2
  - <https://docs.microsoft.com/en-us/windows/win32/xaudio2/xaudio2-introduction>
  - zum Abspielen von Sounds - WIP (nicht implementiert, aber exe linked dagegen)
- DirectWrite
  - <https://docs.microsoft.com/en-us/windows/win32/directwrite/direct-write-portal>
  - zum Rendern von Fonts für Text
- XInput
  - <https://docs.microsoft.com/en-us/windows/win32/xinput/xinput-game-controller-api-portal>
  - Controller Handling - WIP (nicht implementiert, aber exe linked dagegen)