

## Documentation

Hinweis:

Für unser Spiel verwenden wir das ECG-Framework, welches jedoch bis auf die Main-Klasse fast komplett erneuert wurde. Grundlegende Klassen wie zB Shader und Texturen wurden verständnishaft komplett neu geschrieben, teils mithilfe von Tutorials auf Youtube oder Seiten wie <https://learnopengl.com/>

### UPDATE:

- ) Fixing Collision Detection/Bounding Boxes. BB's der Eagles sind jetzt extra-large. Mit jedem Objekt in der Welt kann kollidiert werden.
- ) Versuch die Shadow Map bzgl. Spieler zu verbessern. Das flackern ergibt sich wohl aus dem Modell, aufgrund der dünnen Stäbe.
- ) Refactoring Particle-System, adding „Vulcan“ (At least I tried to create something that looks similar...)
- ) Verschönerung der Umgebung./Mehr Models!!
- ) Fixing der Spieler Movement/Rotation Matrix. Absolute Berechnung → korrekte hierarchische Animation des Propellers

## Compulsory Gameplay:

### Playable

Unser Game ist ein Flugrennsport bei dem es auf Zeit eine bestimmte Anzahl von Checkpoints zu durchfliegen werden müssen. Außerdem gibt es Gegner die nicht berührt werden dürfen, ansonsten verliert der Spieler einen Luftballon. Der Spieler verliert wenn er keine Luftballone mehr hat oder der Timer abläuft bevor er alle Checkpoints durchfliegen hat. Anzahl Checkpoints sowie Timer und Leben werden mittels 2D-Overlay/HUD ingame angezeigt.

Das Durchfliegen und Kollidieren von Gegner basiert auf einer einfachen Collision Detection. Jedes GameObject besitzt eine Bounding Box mit der Möglichkeit diese auf Berührung mit einer anderen Boundingbox eines anderen GameObjects zu testen. Eigentlich gibt es nur einen Checkpoint, der bei Durchfliegen (Collision Detection) auf die nächste Position verschoben wird.

Die Gegner bewegen sich selbständig durch die Spielwelt.

Die Umgebung besteht aus dekorativen GameObjects sowie einem Terrain/Map.

### 3D Geometry

Unsere GameObject Klasse repräsentiert jedes Spielobjekt. Dargestellt werden diese durch Models, welche wiederum, je nach Art Ihrer Erzeugung in GeometryModels(prozedural erzeugte Models) oder ObjectModels (per Model loading Library geladene Models). Erstere ist primär aus dem ECG-Framework übernommen und unterstützt das zeichnen von einfachen Geometry-Objekten wie Cubes oder Spheres. ObjectModels sind komplexe Models die über Zeichenprogramme wie zB Blender erstellt und geladen werden. Dafür bedienen wir uns der Object-Model Library Assimp (<http://assimp.org/>). Der Ladevorgang wurde nach dem Tutorial auf <https://learnopengl.com/Model-Loading/Assimp> implementiert, mit minimalen Änderungen.

### Win/Loose Condition

Wie in Abschnitt „Playable“ bereits erwähnt, gewinnt der Spieler sobald er innerhalb des gegebenen Zeitfensters als Checkpoints durchfliegen hat. Die Loose Condition setzt sich zusammen aus Ablauf des GameTimers oder verlieren aller Leben/Luftballone. Bei Erreichen einer dieser Conditions, wird ein entsprechender Text auf dem Bildschirm ausgegeben.

### Intuitive Controls

In unserem Spiel verwenden wir primär die Tastatur, mittels W/S wird die Flughöhe (pitch) und mittels A/D die horizontale Ausrichtung (yaw) bestimmt. Der Spieler hat eine Mindestgeschwindigkeit, welche mit Leertaste erhöht werden kann (Acceleration based).

Alle Bewegungen sind Framerate independent!

Zusätzlich kann man sich in zwei Axen mittels Maus umsehen: x und y-Achse mittels gedrückter linker/rechter Maustaste

### Intuitive Camera

Wir verwenden eine 3<sup>rd</sup> Person Verfolgungskamera, welche das Spielerobjekt verfolgt und leicht hinter dem Spieler liegend platziert ist. Hierbei gibt es die Möglichkeit sich mittels der Maus in der Umgebung umzusehen. Linker Mausbutton gedrückt in Kombination mit vertikalen Mausbewegungen beeinflusst den Pitch der Kamera zum Spieler. Analog kann mit gedrücktem Rechten Mausbutton in Kombi mit horizontalen Bewegungen der Yaw Wert beeinflusst werden.

### Textures

Mittels des Datei-Pfades werden Texturen mittels einer Image-Loading Library in das Programm geladen. Dafür bedienen wir uns der SOIL2-Library (<https://github.com/alelievr/SOIL2>), welche die bekannte std\_image-Library erweitert und Lademöglichkeiten für alle gängigen Bild-Formate unterstützt. In unserer Texturen-Klasse können wir auch DDS-Formate laden, der Typ wird automatisch erkannt.

### Fonts

Text-Rendering wurde nach dem Tutorial auf <https://learnopengl.com/In-Practice/Text-Rendering> implementiert. Hierbei verwenden wir die Font-Library FreeType (<https://www.freetype.org/>) welche TrueType Fonts lädt und berechnet. TrueType Fonts sind Zeichen die mathematisch definiert sind und mittels Metrics in jeder beliebigen Größe dargestellt werden können ohne Qualitätsverlust. Beim Rendern werden die benötigten Zeichen berechnet, als Texture geladen und auf entsprechend große Rechtecke gerendert. Durch Aneinanderreihung entsteht Text. Zusätzlich unterstützt das Font-Rendering Blending. Hierbei werden Alpha-Werte jedes Pixels verwendet um seine Transparenz zu bestimmen. Dies resultiert darin, dass nur die Zeichen selbst sichtbar sind, und der Hintergrund der Rechtecke komplett transparent ist.

### Moving Objects

In dem Spiel gibt es mehrere gegnerische Spielerobjekte denen es auszuweichen gilt. Diese sind ähnlich dem Spieler aufgebaut, mit dem Unterschied, dass sie sich unabhängig von Eingaben in der Spielwelt bewegen.

### Adjustable Parameters

Zur Einstellung von verschiedenen Parametern wie zB Fenstergröße etc. verwenden wir ein config file, wie es in dem ECG-Framework bereits vorhanden war.

### Illumination/Texturing

Zur Beleuchtung verwenden wir ein einfaches Phong-Shading Model, welches wir aus dem ECG-Framework übernommen haben. Beleuchtet werden alle Objekte.

### Models

Bäume und Luftballons wurden mit Hilfe von Tutorials in Blender erstellt. Die anderen Modelle wurden heruntergeladen und dezimiert bzw. leicht angepasst.

### SkyBox

Wir verwendenen eine Skybox. Hierbei wird die Szene in einen Quader gepackt, der innen mit einer

„Sky“ Texture ausgestattet ist die aufgrund Ihrer Ausrichtung einen Himmel suggeriert. Die Skybox bewegt sich mit der Kamera mit, wodurch sie ingame „still“ steht und starke Größe bzw „Weitsicht“ suggeriert. Realisiert wird diese Skybox mittels einer Cubemap. Einer texture die alle Seiten eines Cubes bedeckt.

## **Optional Gameplay:**

### Heads-Up Display

Wir benutzen ein 2D HUD um den Timer, die Anzahl durchflogener Checkpoints sowie die Anzahl der noch vorhandenen Leben/Luftballone anzuzeigen. Diese Anzeigen/Objekte sind semi-transparent mittels Blending. Dabei wird der Alpha-Channel der Texturen berücksichtigt, der die Transparenz widerspiegelt. Mittels einfachen OpenGL-Befehlen, kann OpenGL angewiesen werden, die Alpha Werte auf eine bestimmte Weise zu berücksichtigen. Wichtig ist hierbei die Render-Reihenfolge der transparenten Objekte. Das HUD kann mittels der Taste F5 ein/ausgeschaltet werden.

### View Frustum Culling

Wir haben View Frustum Culling mithilfe des Tutorials von <http://www.lighthouse3d.com> implementiert. Es werden Objekte, welche außerhalb des Sichtfeldes liegen, nicht gezeichnet, was zu einer Performancesteigerung führen soll. Dabei wurde die Methode gewählt, wo zuerst die Eckpunkte des View Frustums berechnet werden, woraus die Ebenen definiert werden. Von allen Eckpunkten der Bounding Box jedes Objekts im Spiel wird der Abstand zu den Ebenen des View Frustums berechnet. Wenn der Abstand zu einer Ebene positiv ist liegt der Punkt innerhalb davon, ansonsten außerhalb. Sobald ein Eckpunkt der BoundingBox im View Frustum liegt, wird das Objekt gezeichnet, wenn alle Eckpunkte außerhalb liegen, wird es gecullt.

## **Effects:**

### Shadow maps with PCF

Für Shadow-Mapping with PCF hielten wir uns einerseits an die CG-Tutorials, und andererseits an das Tutorial auf <https://learnopengl.com/>.

Bei Shadow-Mapping wird eine Schatten-Textur mit Tiefenwerte generiert, wodurch im Fragment-Shader bestimmt werden kann, ob ein Vertex im Schatten liegt oder belichtet ist.

Zur Erzeugung der Shadow-Map wird die komplette Szene mittels Framebuffer auf eine Texture gerendert, dabei werden nur Tiefenwerte gespeichert. Als Größe der Shadow-Map wählten wir 8k und verwenden diese für unsere gesamte Szene. Als Lichtquelle nutzen wir Directional-Light, das nach NW scheint. Als Antialiasing verwenden wir PCF, bei dem Durchschnittswerte für die Schatten berechnet werden, wodurch die kantigen Schattenränder verschwimmen und so runder wirken. PCF berechnen wir manuell in den Fragment-Shader. Für die Szene exkl. Terrain verwenden wir FrontFaceCulling um ShadowAcne zu vermeiden. Für Das Terrain via Tessellation ist dies nicht möglich, da das Terrain als Plane kein geschlossenes Polygon ist. Hierfür verwenden wir einen PolygonOffset in Kombination mit einer leichten Translation des Terrains nach unten. Tessellation und LOD ist auch beim Shadow-Mapping aktiv und kann hier aufgrund der verändernden, feiner werdenden Schatten aktiv beobachtet werden.

### GPU Particle System using Compute Shader

Das Particle System wurde nach den CG-Tutorials implementiert.

Wir verwenden zwei Shader Storage Buffers, die je die Particle des vorherigen und des aktuellen Frames speichern. Mittels eines Compute-Shaders werden die Particle-Attribute berechnet und den Buffern hinzugefügt. Mit einem weiteren Set an Shadern, zusätzlich mit einem Geometry-Shader, werden aus den Punkt-Partikeln Sprites.

Verwendung findet das Particle-System als Propeller/Motor-Rauchwolke des Spielers. Unser Particle-Emitter sitzt hierbei im Spieler-Objekt beim Propeller und erzeugt abhängig vom Durchschnitt der Framezeit eine Hand voll Partikel (Quads mit Rauch-Textur), die nach kurzer Zeit

verblassen, via Blending.

### Tessellation from height map

Zur Generierung des Terrains wurde Tessellation in Kombination mit einer Heightmap verwendet. Primär hielten wir uns hier nach dem Tutorial auf

<http://ogldev.atspace.co.uk/www/tutorial30/tutorial30.html>; führten jedoch einige Adaptionen und Extensions hinzu.

Für die Tessellation verwenden wir einen Tessellation Control und Evaluation Shader. Der Tess Control Shader bekommt hierbei vom Vertex Shader einen Patch (Gruppe) an Vertices und bestimmt einerseits den Grad der Tessellationlevels und andererseits die Anzahl der ausgegebenen Vertices, in unserem Fall vier, ein Quadrat. Die Berechnung der Tess-Levels berechnen wir dynamisch anhand der Spieler-Position und schaffen hiermit dynamischen LOD. Hierbei kann es vorkommen, dass Löcher entstehen, wenn zwei Patches mit unterschiedlichen Tess-Levels deplatziert werden. Um dies zu verhindern, wird in solchen Fällen der Tess-Level mit 0.5 skaliert, damit er mit dem gegenüberliegenden Patch übereinstimmt.

Im Tessellation Evaluation Shader werden die Attribute der resultierenden Vertices interpoliert und die Vertices selbst in der y-Achse verschoben anhand einer Displacement-Map.

Die dynamische LOD kann ingame mittels F1/"Wireframe"-Mode wunderbar beobachtet werden, einerseits in mittlerer sowie in unmittelbarer Umgebung des Spielers.

Zusätzlich verwenden wir für das Terrain Multitexturing mittels einer BlendMap. Diese gibt den Amount der einzelnen Texturen an, die im Fragment-Shader gemischt werden.

### GPU Vertex Skinning

Vertex Animation wurde nach dem Tutorial auf

<http://ogldev.atspace.co.uk/www/tutorial38/tutorial38.html> implementiert. Hierfür haben wir mehrere Models mittels Blender gerigged bzw ein Skelett verpasst. Dieses lesen wir mittels Assimp ein, traversieren durch die Hierarchie von Knochen und berechnen uns die Knochentransformationen der einzelnen Meshes. Diese übergeben wir dem Vertex-Shader und berechnen uns anhand der gegebenen Knochen und Gewichten jedes einzelnen Vertices die entsprechende Bewegung.

Im Spiel sind die Adler (Flügelschlag) sowie die Eulen (Kopfbewegung) geskinned.

### (Crazy-Propeller:)

Its not a bug, its a feature!

### **Development Status**

Das Spiel ist technisch an sich fertig.

Jedoch wären bei ein paar Punkten noch Anpassungen/Erweiterungen zu machen, zB feinere Collision Detection, Terrainvielfalt/abwechslung. Anpassung von Objekt-scaling etc.

Austausch des Crazy-Propellers gegen einen langweiligen normalen Propeller.

Quellen:

Textur-Baum:

<https://i.imgur.com/31WJ034.png>

Eichhörnchen:

<https://free3d.com/3d-model/squirrel-v2--389774.html>

Propeller:

<https://free3d.com/3d-model/corsair-f4u-propeller-v1--987595.html>

Palette:

<https://free3d.com/3d-model/pallet-334078.html>

Adler:

<https://archive3d.net/?a=download&id=75c6fe22>

Textur-Ring:

<https://cleverhippo.org/chrome-texture>