wipeout

Albert Julian Mayer	0126505	e932	mayer.julian@googlemail.com

Kurzbeschreibung

Das Spiel „wipeout“ ist ein „anti-gravity racing-game“, in dem man nicht nur als erster ins Ziel gelangen muss, sondern auch mit ausgefallenen Waffen oder PowerUps seine Gegner vernichten/behindern oder sich selbst beschleunigen kann.

Hintergrundgeschichte

Wir schreiben das Jahr 2135. Nachdem eine große Chemiefabrik explodiert ist, hat sich eine schlimme Seuche auf der Erde ausgebreitet und den Großteil der Menschheit vernichtet.

König Zeuduck vom Planeten Phrax schickt seine Armee auf die Erde, um den Überlebenden zu helfen. Doch nicht alle Menschen sind den außerirdischen Helfern gegenüber freundlich gesonnen. Besonders eine kleine Gruppe, angeführt vom bösen Cruzyk, will verhindern, dass die Phrakianer auf der Erde bleiben.

Cruzyk hat vor, die Weltherrschaft an sich reißen und fürchtet, die Außerirdischen könnten seine Pläne durchkreuzen.

Cruzyks Truppen überfallen eines Nachts die Phrakianischen Soldaten im Schlaf und töten viele von ihnen.

Zeuduck beschließt, die wenigen Menschen, die noch nicht an der Seuche erkrankt sind, nach Phrax bringen zu lassen. Danach soll die Erde - und mit ihr Cruzyk und seine Anhänger - gesprengt werden.

Doch jemand muss auf die Erde fliegen, um dort die Sprengladung zu zünden.

Zeuducks tapferer Sohn Triphlon meldet sich freiwillig für diese gefährliche Aufgabe. Unglücklicherweise wird er kurz nach seiner Ankunft auf der Erde von Cruzyks Leuten gefangen genommen.

Cruzyk hat sich mittlerweile selbst zum Kaiser der Erde ernannt und lässt zu seiner Belustigung Wettrennen veranstalten. Er hat sich von den Gladiatorenkämpfen im antiken Rom inspirieren lassen und arrangiert eine modernisierte Variante zu seinem Amusement. Es handelt sich dabei um ein Wettrennen um Leben und Tod. Triphlon kommt ihm da gerade recht...

Gameplay

Das Spiel „wipeout“ soll ist vom Gameplay her ein möglichst genauer Klon des Spieleklassikers "WipeOut 2097". Konkret heisst das ein Anti-Gravitations-Rennspiel bei dem auch der bewaffnete Kampf zwischen den Kontrahenten ein wichtiges Element darstellt. Auf drei verschiedenen Rennbahnen wird ein Wettrennen zwischen 9 Kontrahenten ausgetragen. Wer nach 3 Runden zuerst ins Ziel gelangt, hat gewonnen. Der Spieler kann zwischen 5 verschiedenen Vehikeln wählen. Auf der Strecke befinden sich zufällig plazierte "Bonusboxen", deren :

• **Missile**: Ein Missile wird abgefeuert und bremst getroffene Gegner

• **Thunder Bomb**: Eine Bombe wird abgeladen die nageliegende Gegner bremst

• **Quake Disruptor**: Eine "Welle" geht durch die Rennbahn nach vorne und bremst getroffene Gegner

• **Plasma Bolt**: Ein Energieschuss wird abgefeuert und eliminiert getroffene Gegner

• **Speedup**: Das eigene Vehikel wird wird dramatisch beschleunigt - Vorsicht in den Kurven ;->

Objekte im Spiel

• Die Szenerie in der sich die Rennbahn befindet, also Landschaft, Gebäude sowie sich drehende Windräder

• Die Rennbahn (texturiert), inclusive Start/Ziel-gate und Bonusboxen (texturiert)

• Die 5 Anti-Gravity-Vehicles (texturiert)

• Ein Helicopter (texturiert & hierarchisch animiert)

• Die Skybox

• Objekte zur Waffenrepräsentation (Bombe, Missile, Partikelsystem)

Insgesamt besteht die Szene aus zirka 175.000 Dreiecken.

Steuerung

Die für Rennspiele gewohnte Standardsteuerung mit den Pfeiltasten wird verwendet (Alternative: KP4, 8, 6). Achtung, es gibt keine Bremse!

Der Abschuss der Waffen erfolgt mit “Space". Mit „Esc“ wird das Spiel beendet.

Funktionstasten:

F1: Toggle MotionBlur

F2: Toggle FPS-display - Achtung: wenn nicht "Disable SyncToVBL" im Launcher aktiviert, dann max. 60

F3: Toggle WireFrame

F4: Toggle DisplayNormals

F5: Toggle VisualizeOctree

F6: Toggle RenderMode: VBO (Default), VA, DL, Immediate Mode - Achtung, VFC nur bei VBO & VA

F8: Toggle VFC

Beleuchtung

Der ganze Level wird von der Sonne beleuchtet, welche durch eine direktionale Lichtquelle realisiert wird. Die Szene wird mit Nicht-Photorealistischer Per-Pixel-Beleuchtung gerendert.. Diese Lichtquelle wirf auch den Schatten (ShadowMap).

Spezialeffekte

Schon für Echtzeitgraphik implementiert, daher keine Details: Octree für Frustum-Culling, Per-Pixel Gooch-Shading für die Szene, Partikelsystem für Düsenantrieb & Waffen und Motion-blur für High-Speed-Feeling

Für CG2LU implementiert:

• **Schatten mittels ShadowMaps:** Es wurden standard Shadowmaps implementiert, eine 512er für das eigene Vehikel, eine 4096er für alle anderen beweglichen Objekte (Gegner, Bonusboxen, Windmühlen) und eine statische 8192er für die Rennbahnen, diese wird nur einmal berechnet. Benutzte Quellen: Shadowmap Chapters/Samples aus der OpenGL Superbible und dem Orange Book.

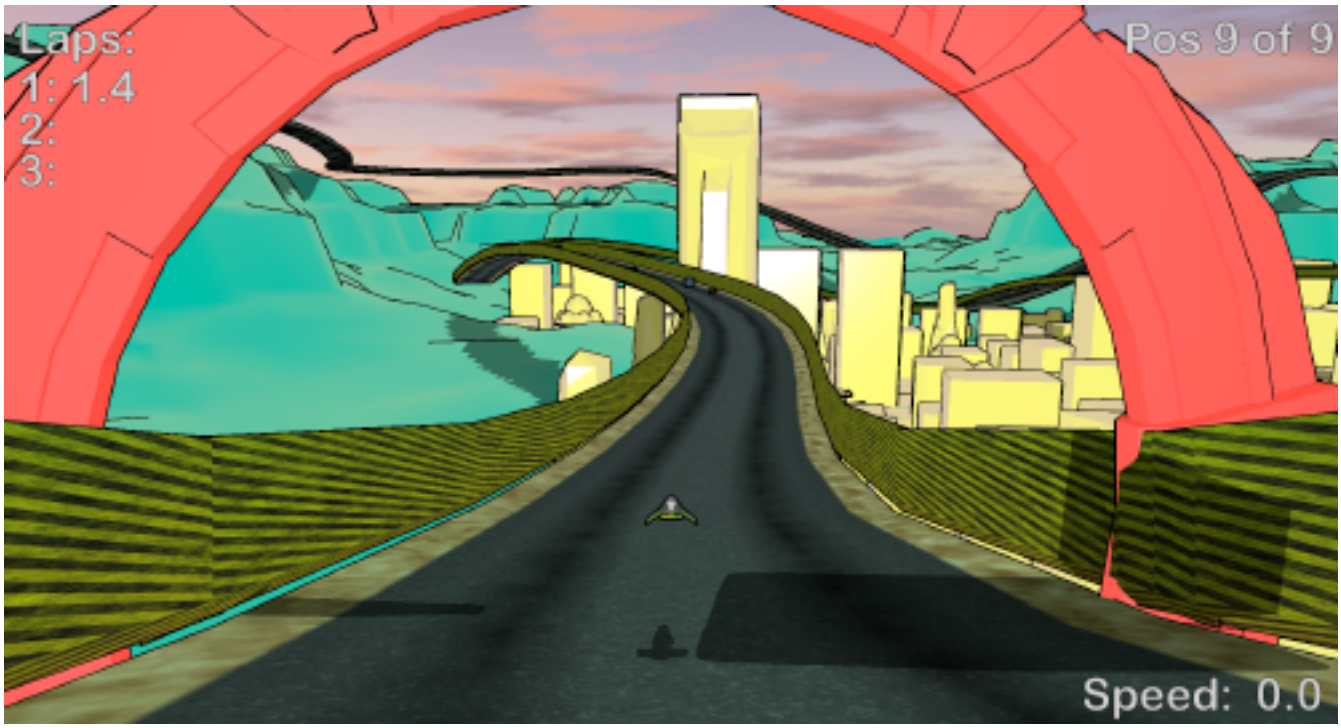
• **Collision-Detection**: Basierend auf der Octree-Datenstruktur wurde eine generische Collision-Detection programmiert die folgende Features bereitstellt:

• intersectOctreeWithLine: Liefert IntersectionPoint
• intersectOctreeWithOctree: Liefert TriangleIntersectionInfo

Folgende Quellen wurden benutzt: OPCODE (SegmentIntersectsAABB), GamaSutra (OBBIntersectsOBB) und "Fast and Robust Triangle-Triangle Overlap Test Using Orientation Predicates" (tri_tri_intersection_test_3d)
• **PowerUp-Effekte:**

- Quake Disruptor: Vertex-Shader hebt und färbt Rennbahn
- Thunder Bomb: Geblendetes Fullscreen-Quad
- Plasma Bolt: Partikelsystem
- Speedup: Dynamische FOV-Anpassung

Screenshot



Weitere Dokumentation:

Da ein grosser Teil des Codes von WipeOut schon für die LVA Echtzeitgraphik entwickelt wurde, ist die damalige Dokumentation noch relevant. Sie beschreibt auch unter anderem die benutzten Libraries, allerdings "FTGL" für das FontRendering hinzugekommen:

DOKUMENTATION:

1.) Allgemeines:

Die diese Demo antreibende "Engine" wurde innerhalb von 2 Monaten speziell für diese und noch eine andere LVA "von Scratch" geschrieben. Die "Engine" wurde in Objective-C entwickelt, performance-relevante Teile auch in "plain C". Die Preprocessing-Tools wurden in Python implementiert. Als Entwicklungsumgebung kam XCode 3.0 unter Mac OS X 10.5 zum Einsatz.

Die bei Objective-C normalerweise benutzen Frameworks "Foundation" und "AppKit" sind leider nur unter Mac OS X (und Nextstep/OpenStep) standardmässig installiert, daher wurde die Library "Cocotron" benutzt, die diese Frameworks auch unter Windows zur Verfügung stellt. Da "Cocotron" nicht perfekt ist, wird auf Win32 zusätzlich "SDL" zum OpenGL-Context-Management und "DevIL" zum Image-Loading benutzt. Eine Implementation für das Sound-Playing wurde im Rahmen dieser LVA entwickelt, dem Cocotron-Projekt beigesteuert und inzwischen in die aktuelle Version aufgenommen.

Zusätzlich wird eine Library namens "OpenGL Mathematics" für Vector-Math benutzt (achtung, es gibt 2 Libraries mit diesem Namen).

2.) "Engine":

Der von der Engine benutzte "Content" wird von den Preprocessing-Tools hergestellt. "generateOctreeFromObj.py" erstellt aus Dateien im Wavefront OBJ File-Format einen optimierten Octree, für direkte Benutzung per VBO und effizientes View-Frustum-Culling. "generatePathFromObj.py" erstellt aus einem OBJ-File mit einer 3D-Linie aus wenigen diskreten Punkten einen cubisch-interpolierten Pfad - dieser wird als Array von Punkten für jeden Frame der Animation gespeichert. In der Engine existiert ein simpler Szenengraph, "SceneNode" implementiert Basisfunktionen wie das Handling der Position/Rotation/ ChildNodes/etc. Der Hauptteil der (Rendering-)Arbeit wird vom "Octree" übernommen, der die optimierte Datei linear in den Speicher einliest und per VBOs und VFC rendert. Diese Classe enthält einige Optionen, wie das Visualisieren der Octree AABBs, das Rendern von Normalvektoren, einen Immediate-Mode Testpfad und die Option zum Front-To-Back Rendering (für Early-Z culling).

Der Engine-Code wurde unter weitgehendem Verzicht auf externe Quellen, Referenzen oder Inspirationen entwickelt - mit Ausnahme von Tutorials zu spezifischen OpenGL-Funktionen.

3.) Demo:

Laut Projektvorschlag soll die Demo ein rasantes Anti-Gravitiy-Race zwischen mehreren Kontrahenten aus der Kameraperspektive eines der Teilnehmer zeigen. Die Modelle für die Szenerie sowie die Gegner wurden dabei bei dem "Ecksdee" Projekt entliehen. Die Szene hat ca. 330.000 Dreiecke.

Die in der Szene enthaltene Kurven für die Rennbahnen wurde als Pfad für die Kamera und die Gegner benutzt. Da es mir leider unmöglich war diese Kurve in Blender anzupassen, bewegen sich die Kamera und die Gegner leider auf dem exakt gleichen Pfad, und zwar mit konstanter Geschwindigkeit - daher kommt beim Betrachten der Demo eher Langeweile als ein Gefühl für ein "rasantes Rennen" auf. Da die Szene und die

Gegner per Gooch-Shading dargestellt werden, kommen keine Texturen zum Einsatz. Die einzigen Texturen stellen die Skybox und die Feuerpartikel dar. Durch das Gooch-Shading und dem damit eingehenden Verzicht auf Texturen kommt leider auch dar Motionblur-Effekt kaum zur Geltung. Gerade auf dem einzigen Objekt mit einer hohen per-pixel-velocity, der Rennbahn, lässt sich das Blurring kaum erkennen. Um den Motionblur-Effekt zu verstärken wurden einige Ringe um die Rennbahn platziert, da ich wiederum unfähig war dies in Blender zu tun, passiert es im Demo-Code.

Bei der Programmierung wurde Wert darauf gelegt das Interface zur Engine möglichst simpel und dennoch "powerful" zu gestalten um den Code für die eigentliche Demo, "Simulation" genannt, klein halten zu können.

4.) Effekte:

Die Effekte wurden alle als Szenengraph-Nodes implementiert z.B. beim GoochShader werden einfach alle Childnodes davon mit Gooch-Shader gezeichnet.

4.1.) Gooch-Shading:

Laut Projektvorschlag: "Non-Photorealistic Rendering using the Gooch matte shading algorithm as outlined in the section "Technical Illustration Example" of the book 'OpenGL Shading Language'"

Gooch-Shading ist ein Nicht-Photorealistisches-Beleuchtungsmodell welches versucht technische Illustrationsmethoden nachzuahmen um Informationen über Form und Struktur dem Benutzer möglichst effizient zu Übermitteln.

Die Implementierung des Gooch-Shading stellte keine Probleme dar. Tatsächlich wurde nur Sample Code dazu angepasst und integriert. Ausserdem wurde mit Toon-shading experimentiert, damit konnte aber bei meiner Szene kein gutes Resultat erzielt werden. Die Implementierung vom Gooch-Shading rendert die gesamte Geometrie wegen der schwarzen Silhouetten doppelt, alternativ könnte man Silhouetten auch per Shader erzeugen.

Folgende Quelle wurde benutzt:

http://developer.apple.com/samplecode/GLSLShowpiece/listing40.html

4.2.) Motion-Blur:

Motion-Blur wurde gemäß dem Projektvorschlag "Motion Blur using the technique presented in "Chapter 27: Motion Blur as a Post-Processing Effect" of the book "GPU Gems 3" implementiert. Diese Technik ist (ohne Erweiterungen) hauptsächlich für statische Szenen mit bewegter Kamera tauglich, was auf meine Demo (abzüglich der Gegner) gut zutrifft. Der Algorithmus berechnet für jeden Pixel die Per-Pixel-Velocity, anhand der View-Projection-Matrix, vom derzeitigen sowie vom letzten Frame - was natürlich nur für statische Objekte korrekt ist. In Richtung dieses Vectors kann dann beliebig oft gesamplet werden. Bei der Implementierung stellten sich folgende Schwierigkeiten: Konvertierung des Shaders nach GLSL inklusive Non-Rectangle Texturing, Beschaffung des Farb- und Tiefenpuffers für die gesamte Szene sowie Herumarbeiten um Treiberfehler. Für die notwenigen Puffer wurden schliesslich sowohl die Möglichkeit der Beschaffung per Rendern der Szene in ein Framebuffer Objekt als auch des Kopierens mit glCopyTexSubImage2D() implementiert. Da dieser Motion-Blur Effekt ein Post-Processing-Effekt ist, das heisst das er auf der fertig gerenderten Szene per Pixelshader operiert, muss er auf die gesamte zu blurrende Szene zugreifen können. Treiberfehler beim Kopieren des Tiefenpuffers per glCopyTexSubImage2D() sowie bei der Schleife im Shader mussten überwunden werden - für das Beschaffen des Tiefenpuffers wird das Voreinstellung auf "FBO" gesetzt, die Schleife wurde manuell für eine fixe Sample-Anzahl "ausgewaltzt". Der Effekt funktioniert gut, ist nur leider in der Demo schlecht zu sehen.

Laut Quellartikel müssen Objekte die nicht geblurrt werden sollen, in unserem Fall also die Gegner, per Maske ausgenommen werden, so dass der Shader diese Pixel ignorieren kann. Hier wurde allerdings die Verbesserung vorgenommen im Shader einfach von der Tiefentextur in den z-Buffer (zurück-zu-)schreiben, sodass Objekte die nicht geblurrt werden sollen einfach nach der Ausführung des Motion-Blur normal gerendert werden können.

Als mögliche Implementierungsverbesserung fällt das Experimentieren mit dem Kopieren des Tiefenpuffers per Renderbuffer ein.

Als Quellen wurden der genannte Artikel im GPU Gems 3 benutzt, sowie ein Tutorial für die FBOs:

http://wiki.delphigl.com/index.php/Tutorial_Framebufferobject

4.3.) Fire-Particlesystem:

Im Projektvorschlag wurde "Fire Rendering using a procedural particle-based method implemented in GLSL" versprochen.

Tatsächlich wurden für das Partikelsystem 3 verschiedene Implementierungen entwickelt, eine CPU-basierte (d.h. ohne GLSL), eine Fragment-Shader-basierte und eine Vertex-Shader basierte version.

Da für den zu erreichenden Effekt keine komplexen Partikelberechnungen notwenig sind wurde es als einfaches "stateless"-Partikelsystem programmiert, eine Implementation im Fragment-Shader wäre sonst auch nicht möglich. Es wurde (noch) nicht versucht generelle Partikelsystemimplemenetationen zu schreiben von denen der spezielle Effekt abgeleitet ist. Ausserdem wurde nicht versucht (physikalisch) realistische Flammensimulation herzustellen, sondern nur die "Düsenantriebe" der Gegner visuell ausreichend darzustellen. Alle 3 Implementierungen rendern per Pointsprites, produzieren aber nicht exakt die gleichen Resultate, z.B. wegen dem Limitationen der Berechnung im Vertex-Shader.

4.3.1) CPU-Version:

Es werden 2 Puffer für Positionen und Geschwindigkeiten initialisiert und jeden Frame die Geschwindigkeit zur Position addiert und tote Partikel neuinitialisiert. Rendern des Positions-arrays.

4.3.2) Vertex-Shader-Version:

Bei der Vertex-Shader-Version wird das Velocity-array gerendert und im Vertex-Shader anhand der Zeit die derzeitige Position extrapoliert. Dabei muss die Startposition für jeden Partikel konstant sein, man könnte aber auch Startpositionen per Textur übergeben.

4.3.3) Fragment-Shader-Version:

Bei der Fragment-Shader-Version werden Positions- und Geschwindigkeitsdaten in eine Textur geschrieben. Jede Textur muss dabei doppelt existieren, da der Fragment-Shader nicht in die aktive Textur schreiben kann und daher Ping-Pong rendering gemacht werden muss. Zuerst lauft nun der Partikelphysik-Fragment-Shader, der per Framebuffer Object aus dem derzeitigen Positions- und Geschwindigkeits-texturen die neuen berechnet/rendert. In unserem Fall wird nicht in die Geschwindigkeits-textur geschrieben, das lässt sich aber mit Multiple Render Targets einfach machen. Dann wird mit dem Partikelrender-Fragment-Shader ein Puffer gerendert der als Vertexpositionen die Indizes in die Positionstextur besitzt. Per Texture Fetch im Vertex-Shader werden die Positionen aus der Positionstextur gelesen und benutzt. Eine andere Möglichkeit die im Frament-Shader berechneten Positionen zu verwenden (neben dem Texture Fetch im Vertex Shader) wäre mittels Pixelbuffer Objects (Render 2 Vertex Buffer). Diese Fragment-Shader-Version funktioniert bei mir wegen Treiberproblemen leider nicht (sobald die Pointsprites aktiviert sind) daher ist sie weder voll funktionsfähig noch ordentlich integriert.

Für die Fragment-Shader version wurde folgende Quelle benutzt:

http://wiki.delphigl.com/index.php/GLSL_Partikel

5.) Fazit:

Wie beim Punkt "Demo" angemerkt gibt es leider ganz offensichtliche Schwächen bei meiner Demo. Da ich jedoch "von Null" gestartet habe und das erste Monat mit dem Basteln der Engine verbracht wurde bin ich mit dem Resultat zufrieden. Ausserdem wurden unter anderem bei dem Partikelsystem und dem Motionblur mehrere Implementierungen entwickelt um zu einem besseren Verständnis zu kommen und Vergleiche ziehen zu können, statt einen Effekt abzuhaken sobald er am Bildschirm erscheint. Persönlich überrascht wurde ich von der Menge an Treiberproblemen die selbst bei so einem kleinen Projekt zu überwinden waren. Ein grosses Problem ist natürlich immer die Content-beschaffung, daher mussten die benutzen Objekte und Szenen anderen Projekten entliehen werden. Es wurde allerdings DEMOnstriert dass sich auch ohne Texturen visuell ansprechende Resultate liefern lassen. Abschliessend könnte man noch sagen dass viel Zeit und einiger unnötiger Code verbraucht wurden um Kompatibilität zu Windows herzustellen, die bei weitem sinnvoller benutzt hätten werden können um die Demo oder die Effekte zu verbessern.