Friss oder stirb - Release 2

Dominik Schörkhuber, 1027470, 033 532
Dominik Dopplinger, 0828648, 033 532

Beschreibung

Friss oder stirb ist ein Gokart-Rennen ohne Gnade. Bis zu sechs Enten jagen sich durch die Arena und schießen sich gegenseitig die Köpfe ein. In einem endlosen Kampf gilt es, die meisten Punkte zu sammeln.

Interaktion

Steuerung

Es werden bis zu vier Game-Pads (XBOX) und zwei Spieler am Keyboard unterstützt:

	Keyboard 1	**Keyboard 2**	**Gamepad**
Beschleunigen	Pfeiltasten	W	Rechter Trigger
Bremsen	Pfeiltasten	S	Linker Trigger
Lenken	Pfeiltasten	A/D	Linker Analog-Stick
Schießen	K	1	A
Handbremse	L	2	B
Bestätigen	Enter	Tabulator	Start

Spielen

Zu Beginn muss sich jeder Spieler durch Betätigen der “Schießen”-Taste registrieren. Für jeden registrierten Spieler wird eine Ente in das Spielfeld gesetzt. Computer-Spieler (Künstliche Intelligenz) können per Leertaste hinzugefügt werden. Das Spiel wird durch die Aktion “Bestätigen” gestartet. (dies ist erst ab zwei Spielern möglich und nur von Spielern, die registriert sind)
Jeder Spieler hat einen Viewport. Bei drei oder fünf Spielern wird der verbleibende Platz für eine Panoramakamera genutzt.

Anmerkungen:

- Das Spiel kann nicht neu gestartet werden. Das Programm muss dazu neu gestartet werden.
- Bälle verschwinden nach dem zehnten Aufprall oder nach fünf Sekunden
- Wenn man umgeworden wird, kann man sich nicht zurücksetzen
- Fünf Treffer töten eine Ente
- Escape beendet das Spiel

Debug-Modus

Während dem Spiel kann durch Drücken von F12 die Debug-Kamera aktiviert werden. Steuerung mit WASD und Pfeiltasten/Mus. Durch F11 wird zum Spiel zurückgewechselt.

Umsetzung der Anforderungen

Gameplay (20)

Wir haben uns dazu entschlossen, keine einsammelbaren Waffen zu entwickeln, da dies mit einem unheimlich hohen Arbeitsaufwand verbunden wäre. Nicht nur die Waffen müssten alle erstellt und gehandelt werden, auch die KI müsste dies berücksichtigen. Deshalb haben wir dem Gameplay der ersten Submission ein Feintuning verpasst, indem wir die Fahrphysik, Waffen und KI verbessert haben. Auch haben wir das Game-Setting aufgrund des hohen Modellierungsaufwandes nciht implementiert und uns auf die Shader-Implementierung konzentriert. Das vorhandene Gamplay reicht jedoch mehr als nur aus, um eine unterhaltsame Beschäftigung zu bieten.

Effects -> Effect-List (20)

Vier Effektpunkte (zwei pro Person) waren vorgeschrieben. Folgende Effekte wurden implementiert:

- **Cel-Shading (0.5 Punkte) (Dominik D.)**
  Offensichtlich. Sichtbar bei den Enten, Pfeilen und Bällen
  Die Farbabstufung ist im "*OpenGL 4.0 Shading Language Cookbook"* beschrieben. Bei unserer Implementierung wird jedoch keine feste Zuweisung verwendet, sondern über Multiplikation, Runden und Division eine Anpassung der Farben erreicht.
  - **+ Contoures (backfaces) (0.5 Punkte) (Dominik D.)**
    Sichtbar an den äußeren Kanten, dem Schwanz und dem Schnabel der Enten. Dieser Shader ist eine Eigenkreation, bei der die back faces im Screen Space um einen festen Abstand gestreckt werden.
  - **+ Contoures (edge detection) (1.0 Punkte) (Dominik D.)**
    Sichtbar an den Farbwechseln innerhalt der Enten. Kanten werden dunkel hervorgehoben
    Der Kantendetektor ist im "*OpenGL 4.0 Shading Language Cookbook"* beschrieben. Bei unserer Implementierung wurde die Kantendetektion jedoch zwei mal durchgeführt, wobei die Gewichtung quadriert wird, um alle Kanten zu

erwischen. Auch die gezeichnete Farbe ist nicht schwarz oder grau. Der vorhandene Farbwert wird über einen Multiplikator verdunkelt.

- **Depth of Field (1.5 Punkte) (Dominik S.)**
  Der Depth of Field Postprocessing Shader benötigt als Input eine Farb und eine Tiefentextur, je nach Höhe des Z-Werts eines Fragments wird dann die Farbtextur stärker oder schwächer mit einem Gaußfilter verwischt. So sind nahe Objekte scharf zu sehen und der Hintergrund der Szene wird immer verschwommener

- **GPU-Particle System(+Transform Feedback,Instancing) (1.0 Punkte) (Dominik S.)**
  Das Partikelsystem ist ein 2 pass shader, wobei der erste pass die vertices bewegt (jedes Partikel wird durch einen Vertex beschrieben), und in unserem Fall die Partikel als Rauch in die Luft steigen lässt. Der zweite Pass ist der eigentliche Rendering Schritt wobei mit einem Geometry Shader jeder Vertex zu einem Billboard für die Rauchtextur erweitert wird und danach mit Alpha Blending rendert.

- **Tesselated Terrain LOD (2.0 Punkte) (Dominik S.)**
  Das Terrain wird je nach Distanz zum Betrachter verschieden stark tesselliert, wobei die Positionen der neuen Vertices mit Phong Tessellation bestimmt werden

  Quellen:
  http://perso.telecom-paristech.fr/~boubek/papers/PhongTessellation/PhongTessellation.pdf
  http://www.swiftless.com/tutorials/opengl/particles.html
  http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html

Complex Objects

Dies ist offensichtlich: Heightmap für die Welt und die Enten-Models. Auch die Partyhats sind importiert.

Animated Objects (10)

Animierte Objekte sind bei den Pfeilen (ehemals Partyhats) zu sehen, die die drei besten Spieler bekommen. Diese bewegen sich im Sinus auf und ab und machen die besten Spieler (und damit besten Punktequellen) sichtbar. Auch die Räder werden von der Spielphysik bewegt.

Frustum Culling (5)

View Frustum Culling wurde implementiert. Die Anzahl der gecullten Frames des ersten Viewports werden im Debug-Output angezeigt.

Transparency

Transparenz wurden im Auspuff-Partikel-System eingebaut.

Controls (5)

Keyboard- und Controller-Inputs wurden für das Spiel implementiert. Die Debug-Kamera kann auch über Maus gesteuert werden.

Experimenting with OpenGL (10) (Transparency, Texture Sampling, MipMapping Quality)

Implementierte Funktionen

Vertex-Buffer-Objects (VBO)

Alle Meshes werden mit Vertex-Buffer-Objects auf der Grafikkarte abgelegt.

Vertex Array Objects (VAO)

Vertex Array Objects speichern die gebundenen VertexBuffer und Attribute Mappings, um beim rendern der Objekte möglichst viele State Changes einzusparen.

Buffer-Objects: FBO (Frame Buffer Object) or UBO (Uniform Buffer Object)

Für das Postprocessing und Shadowmapping werden verschiedene Framebuffer angelegt (Render to Texture) Danach werden die gerenderten Texturen in mehreren Postprocesssing Schritten durch Depth of Field und einem PostProcess Cel Shader verfeinert.

Mip Mapping (on/off)

Für alle geladenen Texturen werden Mipmaps erzeugt, diese können ein und ausgeschalten werden → siehe Funktionstasten.

Textur-Sampling-Quality (Bi/Trilinear Filtering)

Auch die Texture-Sampling-Quality kann per Tastendruck geändert werden → siehe Funktionstasten.

Änderung des Textursamplings und Mipmapping beschränken sich auf geladene tga Texturen, andere Texture Buffer Elemente wie sie z.B. im Postprocessing und Shadowmapping verwendet werden werden dadurch nicht beeinflusst.

Funktionstasten

- F1 - Help
- F2 - Frame Time on/off
- F3 - Wire Frame on/off
- F4 - Textur-Sampling-Quality: Nearest Neighbor/Bilinear
- F5 - Mip Mapping-Quality: Off/Nearest Neighbor/Linear
- F7 - - → lädt alle Texturen und Shader neu
- F8 - Viewfrustum-Culling on/off
- F9 - Transparency on/off

Sonstiges

Anwendung der Shader

Enten, Bälle, Partyhats

Diese Objekte werden mittels eines Cel-Shaders gerendet. Dieser wird zwei mal aufgerufen: Einmal für back faces und einmal für front faces. Der Vertex-Shader berechnet die Position des Objekts am Bildschirm. Der Geometry-Shader unterscheidet zwischen front- und back face. Front faces werden durchgeschleift. back faces werden um einen festen Wert am Bildschirm aufgebläht. Der Fragment Shader unterscheidet zwischen front- und back faces. Back faces werden in schwarz gezeichnet Front faces verwenden einen übergebenen Farb-Vektor, wenden eine Lambert-Beleuchtung darauf an und stufen anschließed über Multiplikation, Rundung und Division die Farbe ab. Die Lambert-Berechnung benutzt einen festen Lichtvektor

Skybox

Der Skybox werden per C++ die Normalen nach dem Laden invertiert. Der Vertexshader berechnet die Position am Bildschirm, lässt die Skybox jedoch immer im selben Abstand zur Kamera darstellen. Der Fragment Shader wendet eine Textur an.

Terrain

Das Terrain wird über eine Heightmap generiert. Der Vertex Shader transformiert das Terrain in den World Space. Der Tesselation Control Shader ändert das Level of Detail je nach Entfernung zur Kamera danach intepoliert der Tessellation Evaluation Shader neue Vertices mit Hlife von Phong Tessellation in den Screen Space. Der Fragment Shader benutzt eine triplanare Texturierung mit einer Lambert-Beleuchtung mit einem festen Lichtvektor.

Postprocess Depth of Field

Anschließend wird ein Depth of Field Shader über das bisher gerendete Bild gelegt, wobei entfernte Objekte weichgezeichnet werden. (Gaussian Blur)

Postprocess Cel Shader

Zum Schluss wird ein Kantendetektor über das neue Bild gelegt. Harte Kanten werden dabei noch betont. Hier werden jedoch keine schwarzen Pixel gezeichnet, sondern vorhandene Farbwerte über einen Multiplikator verdunkelt.

Benutzte Tools

- Blender

Benutzte Bibliotheken

- Assimp: http://assimp.sourceforge.net/
- Boost: http://www.boost.org/
- FMOD: http://www.fmod.org/
- GLEW: http://glew.sourceforge.net/
- GLFW: http://www.glfw.org/
- GLM: http://glm.g-truc.net/

- NVidia PhysX: http://www.nvidia.de/object/physx_new_de.html
- XInput