The Walking Bread – Dokumentation

Gameplay

Der Spieler startet auf einem Terrain, welches von Mauern eingegrenzt ist. Auf dem Terrain befinden sich verschiedene Gebäude (Türme und Häuser). In einem der Häuser (das auf dem Hügel in unmittelbarer Nähe des Startpunkts) befindet sich ein Buzzer. Dieser muss erreicht und aktiviert werden, um das Spiel zu gewinnen.

Der Spieler verfügt über 5 Lebenspunkte. Wenn er all diese Punkte verliert, ist auch das Spiel verloren.

Jede Sekunde fällt ein Brot (=Gegner) vom Himmel, das den Spieler kontinuierlich verfolgt. Erreicht ein Brot den Spieler, so verliert er einen Lebenspunkt. Danach hat der Spieler eine kurze Zeit, in der er keinen Schaden nehmen kann.

Die Gegner können eliminiert werden, indem sie mit Marmelade-Geschossen getroffen werden. Maximal befinden sich 100 Gegner und 100 Geschosse im Spiel. Danach werden die ältesten Geschosse gelöscht, wenn gefeuert wird. Gegner können erst dann wieder spawnen, wenn der Spieler einen Gegner erledigt hat.

Free movable camera

Die Kamera wurde als first person camera implementiert. Mit den WASD-Tasten bewegt sich der Spieler. Die Kamera ändert ihre Position daraufhin, sodass Spielerposition und Kameraposition immer übereinstimmen. Gleichzeitig bestimmt die Blickrichtung der Kamera, in welche Richtung sich der Spieler bewegt. Die Blickrichtung wird mittels Mausbewegung oder mit den Pfeiltasten geändert. Damit der Character bzw. die Kamera nicht auf dem Kopf stehen kann, wurde die Rotation nach oben/unten begrenzt.

Simple lighting and materials

Die Szene wird von direktionalem Licht beleuchtet, welches von schräg oben auf die Szene fällt. Alle Objekte im Spiel (außer das Terrain) haben zurzeit dieselben Materialeigenschaften, die noch im Shader festgelegt sind.

Controls

Neben den Controls, die das Gameplay ermöglichen, sind auch einige zusätzliche Controls eingebaut, die das Debuggen erleichtern.

Gameplay	
Enter	Spiel beginnen
W	Bewegung nach vorne
A	Bewegung nach links
S	Bewegung nach hinten
D	Bewegung nach rechts
E	Buzzer betätigen
Pfeiltaste oben	Nach oben schwenken
Pfeiltaste unten	Nach unten schwenken
Pfeiltaste links	Nach links schwenken

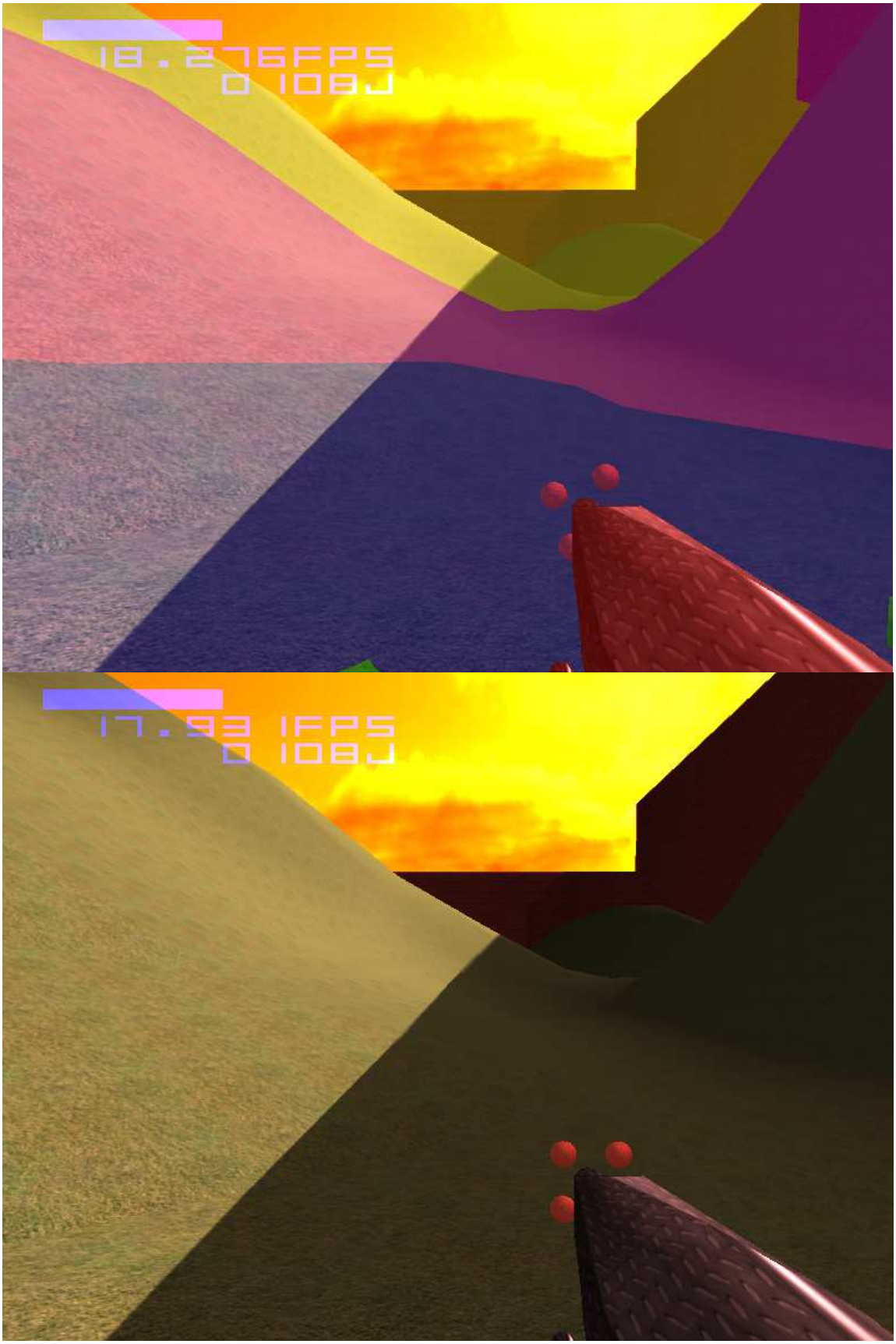
Pfeiltaste rechts	Nach rechts schwenken
Leertaste	Springen
Mausbewegung	Kameraorientierung ändern
Linke Maustaste	Marmelade abfeuern
Esc	Spiel beenden
Debug u.Ä.	
1	Blickrichtung der 1. Shadow Map Cascade
2	Blickrichtung der 2. Shadow Map Cascade
3	Blickrichtung der 3. Shadow Map Cascade
4	Blickrichtung der 4. Shadow Map Cascade
5	Blickrichtung der 5. Shadow Map Cascade
C	Zurück in die Spieler-Blickrichtung wechseln
P	Shadow Map Cascaden Markierung an/aus
U	Spawn der Brote aus/an
I	Ambient Beleuchtung höher
O	Ambient Beleuchtung niedriger
L	Bewegung des Lichts stoppen/starten
F	Bewegung des Lichts vorwärts
B	Bewegung des Lichts rückwärts
OpenGL Experimenting	
F2	Framerate + Gerenderte Objekte an/aus
F3	Wireframe Modus an/aus
F4	Texture Sampling
F5	MipMap Quality
F8	Culling der Brote und Kugeln aus/an
F9	Transparenz aus/an

Effects

Cascaded Shadow Mapping with PCF (2.5 Points)

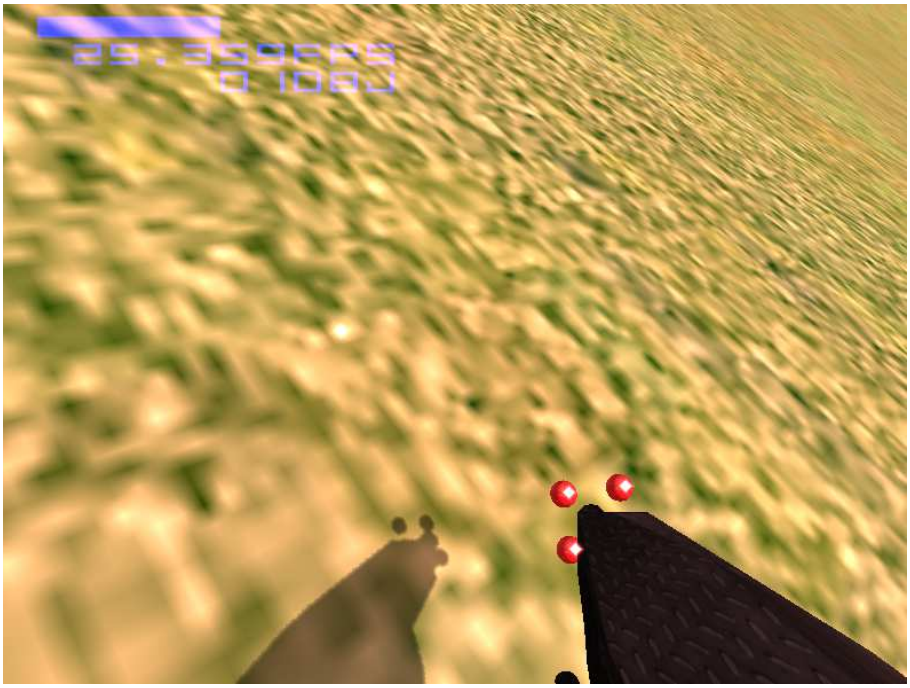
Grundsätzlich funktioniert Cascading Shadow Mapping so, wie es im Vortrag bwz. in den Folien beschrieben wird. Da normales Shadow Mapping für unsere Szene, die sehr groß ist, keine guten Ergebnisse geliefert hat, haben wir Cascaded Shadow Mapping implementiert.

Dazu werden 5 Shadow Maps erstellt, wobei jede Shadow Map unterschiedlich nahe am Spieler gerendert wird. Je nachdem, wie weit ein Objekt von Spieler entfernt ist, wird ermittelt, in welcher Cascade sich das Objekt befindet. Anschließend wird je nach Cascade die passende Shadow Map ermittelt und angewandt. So werden die Schatten der Objekte, die nahe sind, genauer gerendert, wogegen Schatten der Objekte, die weiter entfernt sind, ungenau ausfallen. Das ist aber insofern egal, weil man ohnehin weit vom Objekt entfernt ist und die Auswirkungen der schlechten Schatten nicht sieht. Folgende Abbildungen zeigen die Einteilung der Cascaden und die Auswirkung auf den Schatten:



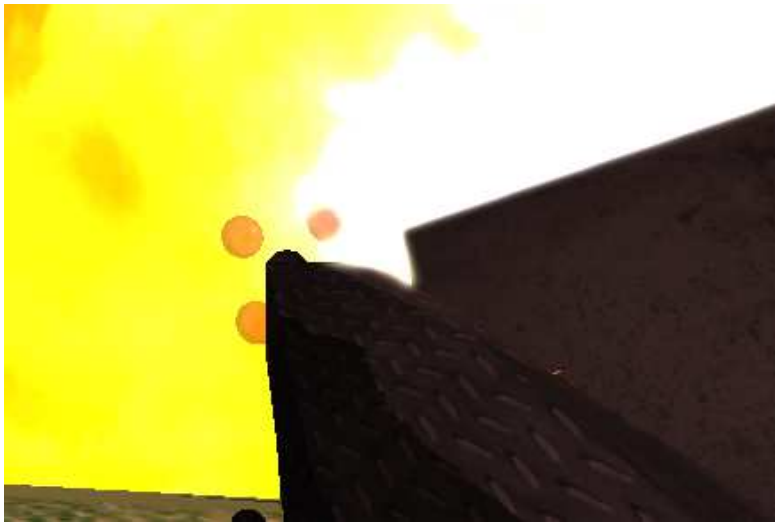
Wie auch bei normalem Shadow Mapping hatten wir mit verschiedenen Artefakten zu kämpfen. Self Shadowing wurde mithilfe eines Bias-Wertes im Shader beseitigt. Peter Panning war kein Problem, da unsere Szene aus „dicken“ Objekten besteht.

PCF wurde mithilfe des OpenGL 4.0 Shading Language Cookbooks (ISBN: 1849514763) implementiert. Dazu werden im Fragment Shader die Werte der Nachbar-Pixel in der Shadow Map addiert und danach gemittelt, was den Wert für die Schattierung ergibt. Dieser Wert wird mit der fragColor multipliziert. Folgende Abbildung zeigt den Einfluss von PCF anhand des Schattens der Waffe:



Bloom (1 Point)

Bloom ist ein Post-Processing-Effekt, bei dem sehr helle Bildbereiche dunklere überstrahlen; das Licht „blutet“ über diese Bereiche. Damit wird ein Effekt simuliert, der z. B. beim Fotografieren entstehen kann. Zur Realisierung werden in mehreren Post-Processing-Schritten die hellen Bereiche extrahiert, mit einem Gauss-Filter verschmiert (man kann hier beispielsweise Mipmaps verwenden, um das Filterergebnis anzunähern und die Performance zu verbessern) und das Ergebnis zum ursprünglichen Bild hinzugefügt. Die Grundzüge unseres Bloom-Effekts wurden mithilfe des OpenGL 4.0 Shading Language Cookbooks (ISBN: 1849514763) implementiert.



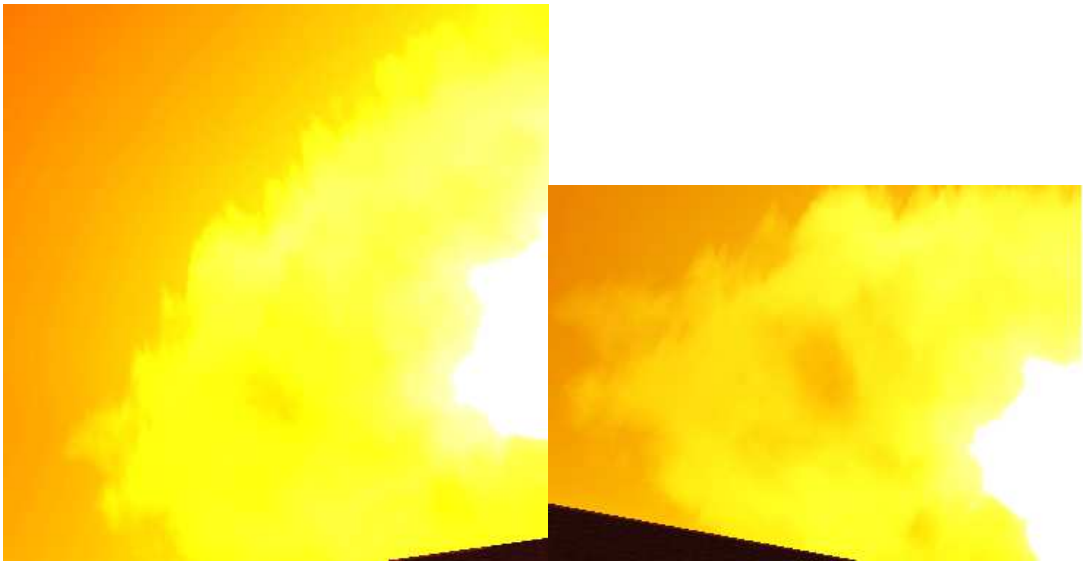
Lens Flare (0.5 Points)

Lens Flares sind Artefakte, die beim Fotografieren aufgrund von Reflexion und Streuung des Lichtes auf der Kameralinse entstehen. Bei Computerspielen werden sie gerne als Post-Processing-Effekt simuliert. Sehr helle Bereiche (Lichtquellen) werden auf die andere Hälfte des Screens gespiegelt. Entlang der Spiegelungsachse werden sogenannte „Ghosts“ erzeugt, die letztendlich zu Lens Flares werden. Dazu werden sie mit einer farbigen Textur kombiniert und Gauss-gefiltert. Die Farbkanale werden verzerrt, was die Lichtbrechung auf der Linse simulieren soll, und anschließend wird das Ergebnis noch mit einer Schmutz-Textur kombiniert, um Schmutz auf der Linse anzudeuten. Lens Flares wurden mithilfe eines Tutorials von John Chapman implementiert (http://john-chapman-graphics.blogspot.co.uk/2013/02/pseudo-lens-flare.html). Die Lens Flares werden nur gerendert, wenn Blickrichtung des Spielers und Lichtrichtung so zueinander stehen, dass die Lichtquelle für den Spieler sichtbar ist.

HDR (1 Point)

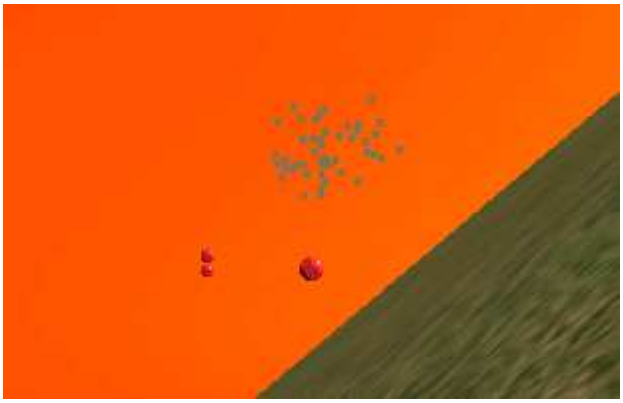
Das menschliche Auge ermöglicht die Wahrnehmung eines viel breiteren Spektrums an Helligkeitsunterschieden, als auf einem Computerbildschirm heutzutage angezeigt werden kann. Um maximale Details in jeder beliebig hellen Szene zeigen zu können, werden bei High Dynamic Range Rendering sämtliche Texturen (=Rendering Targets bei der Verwendung von FBOs) als Floating-Point gespeichert. Um die Texturen letztendlich auf dem Bildschirm anzeigen zu können, benutzt man sogenanntes Tone-Mapping, das die Helligkeitswerte in für das Anzeigen brauchbare Werte zurückkonvertiert. Dabei wird mit einem Exposure-Wert die Lichtaussetzung einer Kamera nachgeahmt und somit die Helligkeit der Szene bestimmt. Damit dennoch möglichst viel Detailreichtum angezeigt wird, haben wir Auto-Exposure implementiert: Der Exposure-Wert ändert sich dynamisch, abhängig von der mittleren Helligkeit der Szene (hierbei werden Mipmaps für schnellere Berechnung verwendet). Für diesen Zweck gibt es mehrere Tone-Mapping-Algorithmen; wir haben einen recht simplen verwendet: Die Helligkeit eines Pixels ist 2^Exposure, wobei Exposure im Verhältnis zur mittleren Helligkeit der Szene steht. Man sieht Auto-Exposure in unserem Spiel

sehr gut, wenn man einen von Schatten verdunkelten Bereich betritt, oder am Himmel, wenn man aus dem (dunkleren) Hausinnenraum ins Freie geht.



Particle System

Unsere Brote zersplattern, wenn man sie abschießt, in primitiven Partikelsystemen, bestehend aus Punkten. Die Größe dieser Punkte verändert sich natürlich mit der Nähe zum Spieler. Die Punkte entstehen zufällig, sind transparent und werden mit der Zeit immer durchlässiger, bis sie ganz verschwunden sind. Position und Transparenz werden dynamisch auf der GPU berechnet. Partikelsysteme wurden unter Zuhilfenahme des OpenGL 4.0 Shading Language Cookbooks (ISBN: 1849514763) implementiert.



Complex Objects

In unserer Szene befinden sich komplexere und konvexe Objekte (Beispielsweise Türme, Häuser, Brote und Terrain). Die Objekte im Spiel wurden mit Blender modelliert, wo sie auch texturiert und mit UV Koordinaten ausgestattet wurden. Lediglich das Terrain wurde nicht selbst modelliert sondern stammt aus dem 3dwarehouse (https://3dwarehouse.sketchup.com/model.html?id=57a13fbe921e0cf7a9052d3360a66fd6).

Animated Objects

Das animierte Objekt in unserem Spiel ist die Ansammlung an Kugeln, die um die Waffe des Spielers kreisen. Die Waffe und die 4 Kugeln werden aus jeweils zwei separaten Collada Files eingelesen. Die Kugeln werden zur Laufzeit so positioniert, dass sie sich vorne am Lauf der Waffe befinden und darum herum kreisen.

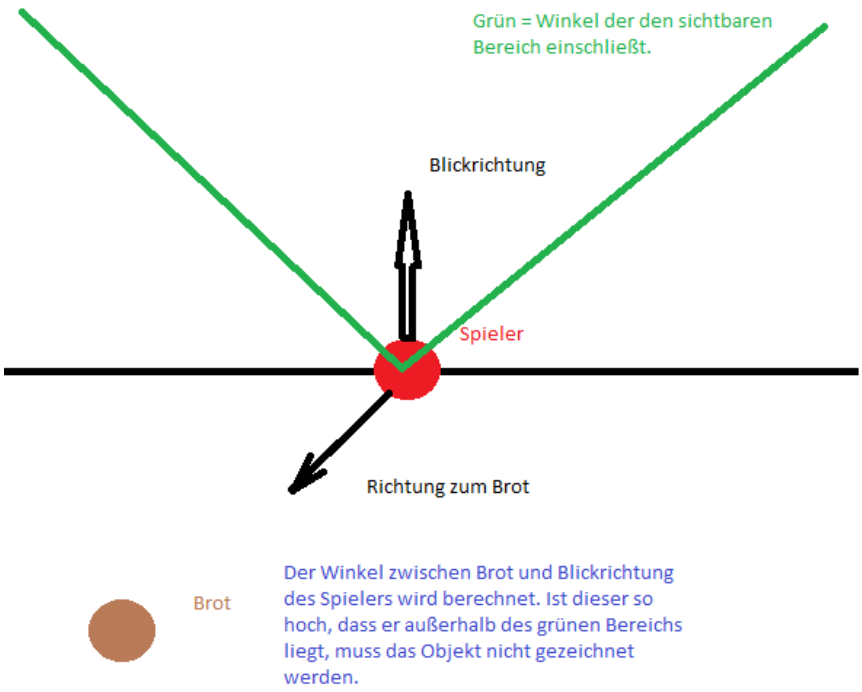
Frustum Culling

Für die Implementierung des Cullings wurde folgende Überlegung angewandt:

Der sichtbare Bereich wird grundsätzlich vom Öffnungswinkel der Kamera bestimmt. Alles, was außerhalb des Öffnungswinkels liegt ist nicht sichtbar und muss daher auch nicht gezeichnet werden.

Da die Brote und die Geschosse die einzigen dynamischen Objekte im Spiel sind, die gleichzeitig sehr häufig auftreten können, werden sie gecullt.

Dazu werden 2 Vektoren betrachtet: Die Blickrichtung des Spielers und der Vektor vom Spieler hin zum Geschoss/Brot. Diese Vektoren werden auf die xz-Ebene gelegt und anschließend der Winkel zwischen den Vektoren berechnet. Ist der Winkel so groß, dass er außerhalb des Öffnungswinkels fällt, so wird das Objekt nicht gezeichnet. Folgende Skizze soll die Überlegung veranschaulichen:

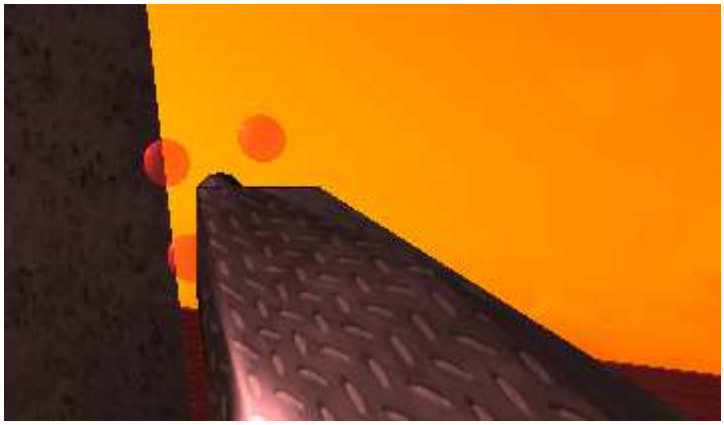


Experimenting with OpenGL

Transparency

Wir haben uns entschieden, die Kugeln, welche die Waffe umkreisen, transparent zu gestalten. Dazu wurde ein eigener Shader implementiert, welcher den Alpha-Wert der fragColor auf 0.3 setzt. Zusätzlich dazu musste mit dem Befehl glEnable(GL_BLEND) Blending aktiviert und danach die glBlendFunc-Funktion

gesetzt werden. Nach dem Zeichen wird das Blending wieder deaktiviert. Die Kugeln selbst werden an vorletzter Position gerendert (die Waffe wird am Ende gerendert).



Texture Sampling & MipMapping Quality

Die Sampling-Qualität von Texturen und die Berechnung der Mipmaps wird in OpenGL anhand des Befehls glTexParameteri(…) eingestellt. Mittels F4 (Textur-Sampling-Qualität) und F5 (Mipmapping-Qualität) können diese Einstellungen geändert werden.

Wireframe Mode

Das Rendern als Wireframes kann mit dem Befehl glPolygonMode(…) eingestellt werden. Der Wireframe-Modus wird in unserem Spiel mit F3 aktiviert. Da wir FBOs und Screen-filling Quads benutzen, werden diese im Wireframe-Modus deaktiviert, damit man wirklich die „richtigen“ Objekte sieht. Post-Processing-Effekte wie Bloom und Lens Flares brauchen im Wireframe-Modus ohnehin nicht angezeigt zu werden.

Additional libraries

Assimp http://assimp.sourceforge.net/
DevIL http://openil.sourceforge.net/
GLEW http://glew.sourceforge.net/
GLFW http://www.glfw.org/download.html
PhysX https://developer.nvidia.com/physx-sdk
GLM http://glm.g-truc.net/0.9.5/index.html
IrrKlang http://www.ambiera.com/irrklang/downloads.html

Code References

Beleuchtung

Für die Beleuchtung wird derzeit für alle Objekte der selbe perFragment

Shader verwendet (Ausnahme: Skybox, deren Shader lediglich die Textur aufträgt, und Terrain, das Gras darstellen soll und dessen Shader daher keine Specular Highlights berechnet)
Für diesen Shader wurde als Hilfe folgendes Tutorial herangezogen:

http://www.lighthouse3d.com/tutorials/glsl-tutorial/directional-light-per-pixel/

Kamera

Für die Implementierung der Kamera war folgendes Tutorial sehr hilfreich:

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-6-keyboard-and-mouse/

PhysX

Für den Einbau der Physik waren folgende Nachschlagewerke hilfreich:

Generelles Nachschlagewerk für PhysX

https://developer.nvidia.com/sites/default/files/akamai/physx/Index.html

Praktisches PhysX Tutorial (besonders Bouncing Box Tutorial)

http://mmmovania.blogspot.co.at/search/label/PhysX3%20Tutorials

PhysX API Reference

http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/apireference/files/main.html

Lens Flares

Wie erwähnt war zum Erzeugen des Lens Flares dieses Tutorial sehr nützlich:
http://john-chapman-graphics.blogspot.co.uk/2013/02/pseudo-lens-flare.html

Framebuffer Objects

Für das Kennenlernen von FBOs, Screen-Filling Quads und Post-Processing-Effekten war dieses Tutorial nützlich:

http://antongerdelan.net/opengl/framebuffers.html

Shadow Maps

Für die Implementierung der Shadow Maps wurde wie in der Vorlesung und in den Folien beschrieben vorgegangen. Zusätzlich dazu war folgender Link hilfreich, der

die Vorgangsweise ebenfalls erläutert (speziell die Idee mit dem Bias wurde anhand der Quelle angewandt):

http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

Cascaded Shadow Maps

Folgende Quellen waren für das Grundverständnis von Cascaded Shadow Maps hilfreich:

http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf

http://msdn.microsoft.com/en-us/library/windows/desktop/ee416307(v=vs.85).aspx

PCF

PCF wurde mithilfe des OpenGL 4.0 Shading Language Cookbooks (ISBN: 1849514763) implementiert.

Irrklang

Musik und Sound wurden so implementiert, wie es in den Tutorials nachzulesen war:

http://www.ambiera.com/irrklang/tutorial-helloworld.html