Censor Everything!

20.06.2016

Markus Klein, 1426483

Timon Höbert, 1427936

Censor Everything!

Gameplay & Physik

„Censor Everything!“ ist ein Action/Adventure-Spiel, in dem man als Oberhaupt eines diktatorischen Staates alles zensieren muss, das nicht regelkonform ist. So muss zum Beispiel jede Form von Meinungsfreiheit in Zeitungen und TV unterbunden werden und jegliche Nacktheit bedeckt werden.

Der Spieler steuert das Staatsoberhaupt und muss alle “bösen” Objekte finden und zensieren. Diese Objekte sind zB Fernseher, Zeitungen, nackte Statuen, usw.

Gesteuert wird das Spiel mittels Maus zum Bewegen der Kamera und über W,A,S,D für Bewegungen des Protagonisten. Mittels Klick können schwarze Zensurblöcke abgefeuert werden, welche beim Auftreffen auf böse Objekte diese zensieren. Dabei werden bei Kollisionen von den Zensurboxen mit den zensierbaren Objekten beide Objekte entfernt und anschließend nur das zensierte Objekt angezeigt. Mittels Leertaste kann man außerdem springen.

Die Zensurblöcke wie auch alle anderen dynamischen Objekte bewegen sich mithilfe der Nvidia PhysX Library physikalisch korrekt im Raum. Dabei wirken alle Objekte aufeinander ein (Kollisionen & Bewegungen), sowie auch die Erdbeschleunigung. Dies wirkt auch auf den gesteuerten Protagonisten ein, welcher zusätzlich auch automatisch über kleinere Objekte wandert (Autostepping).

Komplexe Objekte

Die Meshes von Objekten werden mittels Open Asset Import Library (Assimp) geladen. Im folgenden Auszug wird dabei die Scene aus dem File geladen, und über angenäherte Triangles weiterverarbeitet.

```
const aiScene* pScene = Importer.ReadFile(Filename.c_str(), aiProcess_Triangulate |
aiProcess_GenSmoothNormals | aiProcess_FindDegenerates);
loadSuccess = loadScene(pScene, Filename);
```

Anschließend müssen die Referenznummern von Material, Indizes, Vertex von jedem Mesh gespeichert werden.

```
for (unsigned int i = 0 ; i < mashes.size() ; i++) {
  mashes[i].MaterialIndex = pScene->mMeshes[i]->mMaterialIndex;
  mashes[i].NumIndices = pScene->mMeshes[i]->mNumFaces * 3;
  mashes[i].BaseVertex = NumVertices;
  mashes[i].BaseIndex = NumIndices;
  NumVertices += pScene->mMeshes[i]->mNumVertices;
  NumIndices  += mashes[i].NumIndices;
}
```

Bei jedem Mesh müssen anschließend die Vertexdaten gespeichert werden. Dazu müssen außerdem die Normalen sowie die Texturkoordinaten gespeichert werden.

```
for (unsigned int i = 0 ; i < paiMesh->mNumVertices ; i++) {
        const aiVector3D* pPos      = &(paiMesh->mVertices[i]);
        const aiVector3D* pNormal   = &(paiMesh->mNormals[i]);
        const aiVector3D* pTexCoord = paiMesh->HasTextureCoords(0) ?
              &(paiMesh->mTextureCoords[0][i]) : &Zero3D;

        Positions.push_back(vec3(pPos->x, pPos->y, pPos->z));
        Normals.push_back(vec3(pNormal->x, pNormal->y, pNormal->z));
        TexCoords.push_back(vec2(pTexCoord->x, pTexCoord->y));
}
```

Zum Modellieren der Objekte haben wir Blender verwendet. Es hat zwar nicht immer so funktioniert, wie man es sich vorgestellt hat, jedoch hat es seinen Zweck erfüllt. Vor allem haben wir die Boolean-Operatoren verwendet, um komplexe Objekte aus den Quadern zu schaffen. Dies erzeugte auch die gewünschte rechteckige Optik. Außerdem wurde die Bevel-Funktion angewandt, um Fehler bei der Beleuchtung zu verhindern, die durch falschstehende Normalen erzeugt werden könnten.

Als Container-Objekt für alle Objekte und Lichter in der Spielwelt gibt es einen Szenengraph, der die verschiedenen Objekte in mehreren Maps mit ihrem einmaligen Namen aufbewahrt.

Animierte Objekte

Um komplexe Objekte relativ zueinander zu animieren, wurde ein DerivedObject implementiert, welches relative Positionierung eines Objektes zu einem anderen erlaubt. Dadurch wird es beispielsweise möglich, dass sich der Arm des Partymenschen wie nachfolgend illustriert relativ zum restlichen Körper bewegt. Zusätzlich wird das ganze Objekt, also Körper plus Arm zusammenhängend im Raum physikalisch korrekt transformiert. Dies wird ermöglicht indem das relative Koordinatensystem des Armes mit dem Koordinatensystem des Körpers multipliziert wird.

View-Frustum Culling

Um die Anzahl der gerenderten Objekte pro Bild gering zu halten, verwenden wir View-Frustum Culling. Dabei kommen 6 Ebenen zum Einsatz, deren Positionierung bei jedem Frame neu berechnet werden. Jede dieser Planes wird durch einen Punkt einen Normalvektor definiert, wobei der Vektor stets in das View-Frustum hineinzeigen muss.

Zur Berechnung der Ebenen verwenden wir die Eckpunkte der Near-Plane und der Far-Plane. Diese können mithilfe der near- & far-Distanz, dem vertikalen Öffnungswinkel (Field Of View), dem Seitenverhältnis des Fensters und den drei Richtungsvektoren front, up und right der Kamera wie folgt berechnet werden:

```
//calculate plane measures
float heightNear = 2.0f * tan(fov / 2.0f) * _nearFar.x;
float widthNear = heightNear * _aspectRatio;

float heightFar = 2.0f * tan(fov / 2.0f) * _nearFar.y;
float widthFar = heightFar * _aspectRatio;

//calculate bounding points
vec3 nm  = camPos + camDir * _nearFar.x;
vec3 ntl = nm + (camUp * heightNear / 2.0f) - (camRight * widthNear / 2.0f);
vec3 nbl = nm - (camUp * heightNear / 2.0f) - (camRight * widthNear / 2.0f);
vec3 nbr = nm - (camUp * heightNear / 2.0f) + (camRight * widthNear / 2.0f);

vec3 fm = camPos + camDir * _nearFar.y;
vec3 ftl = fm + (camUp * heightFar / 2.0f) - (camRight * widthFar / 2.0f);
vec3 fbl = fm - (camUp * heightFar / 2.0f) - (camRight * widthFar / 2.0f);
vec3 fbr = fm - (camUp * heightFar / 2.0f) + (camRight * widthFar / 2.0f);

vec3 ntl_ftl = ftl - ntl;
vec3 nbl_fbl = fbl - nbl;
vec3 nbr_fbr = fbr - nbr;
```

Sobald das View-Frustum korrekt konstruiert wurde, muss jedes Objekt auf Sichtbarkeit überprüft werden. Hierbei kommen Boundingboxen zum Einsatz, die

beim Laden der Objekte erzeugt werden. Diese Boxen bestehen aus 8 Eckpunkten und werden durch die minimalen/maximalen Vertex-Koordinatenwerte konstruiert.

Bevor ein Objekt gerendert wird, wir überprüft, ob es innerhalb des View-Frustums liegt, ansonsten wird es nicht gezeichnet. Besser gesagt, wird getestet, ob es eine Ebene gibt, bei der alle Punkte der Boundingbox außerhalb liegen. Ansonsten hatten wir das Problem, dass große Objekte wie das ganze Gangsystem nicht gezeichnet werden, obwohl man in der Mitte davon stand.

Die Überprüfung wurde für jedes Objekt folgendermaßen implementiert:

```
bool ViewFrustumCuller::objectInside(vector<vec4> boundingBox) {
	bool inside = true;

	for (unsigned int i = 0; i < _planes.size() && inside; i++) {
		bool pointInside = false;

		//if one point is inside of plane, we accept it for this plane
		for (unsigned int j = 0; j < boundingBox.size() && !pointInside; j++) {
			if (_planes.at(i)->isInside(vec3(boundingBox.at(j)))) {
				pointInside = true;
			}
		}

		//if all points are outside of one plane, it is rejected
		if (!pointInside) {
			inside = false;
		}
	}
	return inside;
}
```

Tutorials:

- http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes/

Transparenz

In jedem realen Wohnblock gibt es irgendwelche durchsichtigen, transparenten Objekte, so auch in unserem. Transparente Objekte sind in unserem Fall einige Weinflaschen, die in verschiedensten Wohnungen verteilt sind.

Die Transparenz eines Objektes ist in der zugehörigen mtl-Datei definiert worden und gilt stets für das gesamte Objekt. Außerdem wurde das OpenGL Alpha-Blending aktiviert. Um transparente Objekte korrekt darzustellen, müssen sie nach den opaken Gegenständen gerendert werden. Dazu gibt es im Szenengraph-Objekt die Möglichkeit, undurchsichtige und transparente Objekte getrennt zu zeichnen und somit die durchsichtigen Gegenstände zum Schluss zu rendern.

OpenGL-Funktionalität

Um einige Prozesse zu beschleunigen oder zu erleichtern, werden in unserem Spiel auch einige Funktionen von OpenGL ausgenützt. So verwenden wir zum Speichern der Objekte mehrere VBOs (Vertex Buffer Object) zum Aufbewahren von Positionen, Normalen, UV-Koordinaten und Tangenten direkt auf der GPU. Um die VBOs zusammenzufassen, benutzen wir außerdem ein VAO (Vertex Array Object) pro Modell.

Für das Erzeugen von Schatten und die Nachbearbeitung von gerenderten Szenen verwenden wir FBOs (Framebuffer Object), um nicht direkt ins Fenster, sondern in eine Textur zu rendern, die wir danach weiterverarbeiten können.

Außerdem können bei uns verschiedene Textursampling-Methoden ausgewählt werden. Durch einen Tastendruck (siehe HowTo) kann zwischen gewählt werden, ob eine Texturfarbe durch das nächste Pixel oder doch durch eine Interpolation der umliegenden Pixel bestimmt werden soll.

Ebenso kann Mip-Mapping durch einen Tastendruck eingeschalten und durch verschiedene Methoden gewechselt werden.

Features

Beleuchtung

Zur Beleuchtung benutzen wir ambientes Licht, mehrere Punktlichter und auch mehrere Spots. In jedem Raum befindet sich mindestens ein Punktlicht oder Spot, um diffuse Beleuchtung und Glanzpunkte zu erzeugen.

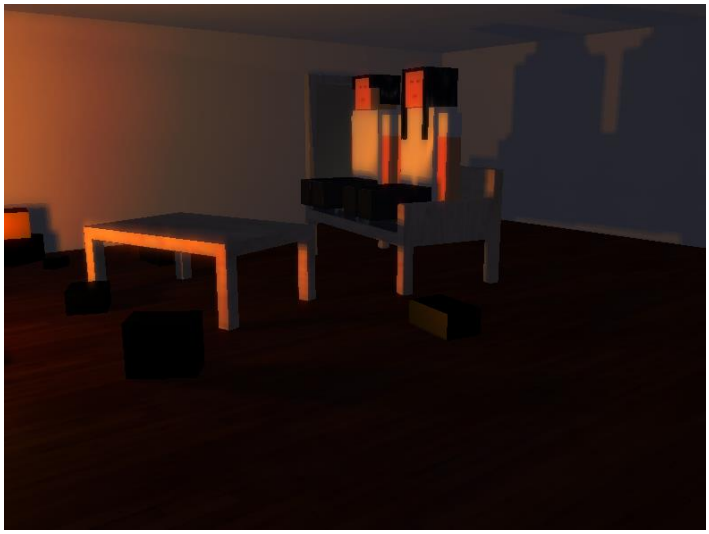
Um die Beleuchtung etwas interessanter zu gestalten, haben wir für einige Lichter besondere Funktionalität hinzugefügt. So gibt es zum Beispiel in einem Raum ein

Discolicht, also ein Spotlight, das immer wieder Farbe und Richtung wechselt. Ein anderer Spot stellt ein flackerndes Licht am Gang dar, er wird zufällig ein- oder ausgeschalten. Im zweiten Stock gibt es außerdem einen Kamin, bei dem durch ein sich drehendes Punktlicht die Bewegung eines Feuers imitiert werden soll.

Effekte

Omni-Directional Shadow Mapping

Gerade in einer Spielwelt wie unseren, wo das Licht sehr stark auf kleine Räume begrenzt ist, ist es besonders wichtig, Lichtstrahlen zu begrenzen und somit Schatten zu erzeugen. Wir haben uns für den ungerichteten Schattenwurf und ihn mithilfe einer Cube Map implementiert.



Im Detail gibt es ein FBO, das in eine Cube Map für die 6 Seiten ein Tiefenbild rendert. Um die Zahl der Renderprozesse zu senken, verwenden wir außerdem einen Geometry Shader, mit welchem wir die verschiedenen Richtungen in einem Durchlauf zeichnen können.

Im normalen Renderprozess wird diese Cube Map dann dazu verwendet, das am nächsten zur Lichtquelle ist, zu beleuchten und Objekte, deren Distanz größer als im Tiefenbild ist, nicht zu beleuchten und somit in den Schatten zu stellen.

Wie folgt wird überprüft, ob der aktuelle Abstand zur Lichtquelle mehr ist als die in der Cube Map gespeicherte Distanz:

```
//omnidirectional shadowMapping
float shadow = 0.0;
float eps = 0.15;
float diskRadius = 0.05;
```

```
vec3 direction = v_position - pointLights[0].v_position;
float depth_p = length(direction);
float depth_c;
for(int i = 0; i < numSampleDir; i++)
{
	depth_c = texture(i_shadowMapcubeUnit, direction +
				cubeShadowSamplingDirection[i] * diskRadius).r;
	depth_c = depth_c * v_nearFar.y;
	if(depth_c+eps < depth_p) shadow += 1.0;
}
shadow /= float(numSampleDir);
shadowFactor = 1.0 - shadow;
```

eps dient als kleiner Spielraum, um „Shadow Acne“ zu verhindern. Die SamplingDirections werden verwendet, um Schattenkanten zu glätten (PCF).

Um die Anzahl der gerenderten Objekte auch hier zu begrenzen, haben wir ein besonderes View-Frustum implementiert, das einen Quader um einen Punkt mit einer bestimmten horizontalen Weite und einer spezifischen Höhe hat, um die Objekte, die für den Schattenwurf in Frage kommen, auf ein Stockwerk zu beschränken.

Tutorials:

- https://www.cg.tuwien.ac.at/courses/Realtime/repetitorium/VU.WS.2013/OmnidirShadows.pdf
- http://learnopengl.com/#!Advanced-Lighting/Shadows/Point-Shadows

Normal Mapping

Bei Normal Mapping, auch als Bump Mapping bezeichnet, werden die derzeit interpolierten aber dennoch glatten Oberflächennormalen eines Objektes abhängig vom Material ähnlich einer Texturierung spezifiziert. Dadurch können raue Oberflächen von Objekten wie Stein genauer modelliert werden. Wie im nachfolgenden Screenshot ersichtlich wird das Licht an der Oberfläche des Steinofens diffus abgestrahlt, wodurch ein räumlicher Eindruck entsteht.

Zur Spezifikation der Normalen wird eine normale RGB-Textur, die Normal-Map eingebunden, wo die RGB-Komponenten den dreidimensionalen Normalvektor beschreiben. Dadurch dass die meisten Normalen fast rechtwinkelig stehen, entsteht der Blaustich des Bildes. Für das Mapping der Koordinaten berechnet man die Tangente (tangent) und deren normale (bittangent), welche den Tangentenraum aufspannen. Diese Berechnung wird mittels *aiProcess_Triangulate* von der ASSIMP-Library bei jedem Vertex automatisch durchgeführt, der Zugriff funktioniert analog zu den Normalen. Im Shader wird anschließend wie folgt die modifizierte Normale berechnet:

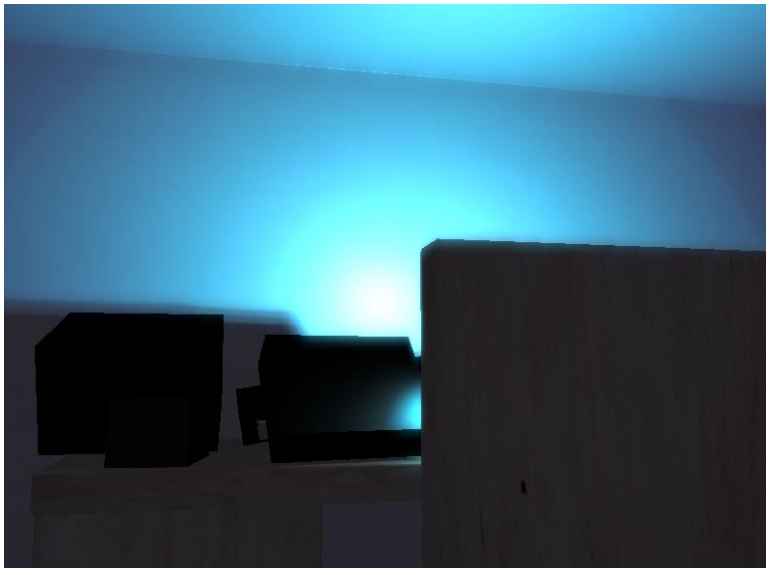
```
vec3 CalcBumpedNormal()
{
    vec3 Normal = normalize(v_worldNormal);
    vec3 Tangent = normalize(v_tangent);
    Tangent = normalize(Tangent - dot(Tangent, Normal) * Normal);
    vec3 Bitangent = cross(Tangent, Normal);
    vec3 BumpMapNormal = texture(i_normalUnit, fragmentUV).xyz;
    BumpMapNormal = 2.0 * BumpMapNormal - vec3(1.0, 1.0, 1.0);
    vec3 NewNormal;
    mat3 TBN = mat3(Tangent, Bitangent, Normal);
    NewNormal = TBN * BumpMapNormal;
    NewNormal = normalize(NewNormal);
    return NewNormal;
}
```

Tutorials:

- http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

Bloom

Um die etwas getrübte bzw. beeinträchtigte Sicht eines Diktators passend darzustellen, haben wir uns für den Bloom-Effekt oder auch Lightbleeding Effekt entschieden. Hierbei werden besonders helle Bereiche im Post Processing geglättet und somit verschwimmen helle Bereiche über die Kanten zu dunklen Bereichen.



Um Renderprozesse zu sparen, zeichnen wir die gesamte Spielwelt in ein FBO mit zwei Color Attachments. Das erste beinhaltet die normal gerenderte Welt, das zweite enthält nur noch Farben, die Größe als ein gewisser Offset (Bei uns ist dies ein drei dimensionaler Vektor vec3(0.3, 0.25, 0.25)) sind.

Danach wird das Bright Pass Bild auf ein Viertel seiner Größe skaliert und zweimal durch einen Gauß-Filter mit einer Kerngröße von 9 Pixeln geschickt, einmal horizontal und einmal vertikal.

Zum Abschluss werden die normal gerenderte Szene und das geglättete Helligkeitsbild addiert und ins Fenster gerendert.

Tutorial:

- http://learnopengl.com/#!Advanced-Lighting/Bloom

Spotlights

Um das begrenzte Licht im Wohnblock zu erzeugen, haben wir zuletzt noch Spots implementiert. Diese stellen quasi eine Kombination von Punkt- und gerichtetem Licht dar. Zusätzlich haben sie einen Öffnungswinkel, der sie noch ein bisschen mehr einschränkt.

Die Berechnung der Spots sieht im Fragment Shader folgendermaßen aus:

```
vec3 L = normalize(spotLights[i].v_position - v_position);
float spotFactor = dot(L, spotLights[i].v_direction);

if(spotFactor > spotLights[i].f_cutoff) { … }
```

Zuerst werden alle Punkte ausgeschlossen, deren Winkel zur Richtung größer als der erlaubte Öffnungswinkel ist. Danach erfolgt eine ähnliche Lichtberechnung wie bei Punktlichtern.

Um harte Kanten an den Rändern zu verhindern, wird der Bereich innerhalb des Beleuchtungskegels noch mit einem Faktor abgeschwächt:

```
//mapped from [cutoff, 1] to [0,1]
float radialIntensity = (1.0 - (1.0 - spotFactor) * 1.0/(1.0 -
                                                    spotLights[i].f_cutoff));

finalColor += shadowFactor * lightColor * radialIntensity;
```

Tutorial:

- http://ogldev.atspace.co.uk/www/tutorial21/tutorial21.html

How To

Die Steuerung des Spieles ist wie folgt:

- WASD zum Bewegen des Spielers
- Mausfahren zum Schwenken der Kamera
- Leertaste zum Springen
- Linke Maustaste zum Schießen eines Zensurbalken

Weiters werden folgende Tasten zum Debugging verwendet:

- F2 Frametime an/aus
- F3 Gitternetz an/aus
- F4 Texturesampling Qualität Nearest Neighbor/Bilinear
- F5 Mip Mapping: Nearest Neighbor/Linear/aus
- F7 Effekte an/aus (Bloom und Shadow Mapping)
- F8 View-Frustum Culling an/aus
- F9 Transparenz an/aus
- ESC Vorzeitiges Beenden des Spieles
- TAB Vortäuschen, dass alle Objekte zensiert wurden

Das Ziel des Spieles ist es 13 Objekte zu zensieren. Um das Spiel zu gewinnen, muss man nur noch in die Küche der reichen Leute gehen (Wohnung mit Statuen). Bei grünem Licht gewinnt man und das Spiel beendet nach 5 Sekunden, bei rotem Licht muss man weitersuchen.

Die Fenstergröße kann in der „config.txt“ Datei gesetzt werden. Hierbei gelten folgende Werte:

[width] [height] [fullscreen on(1)/off(0)] [refresh rate]