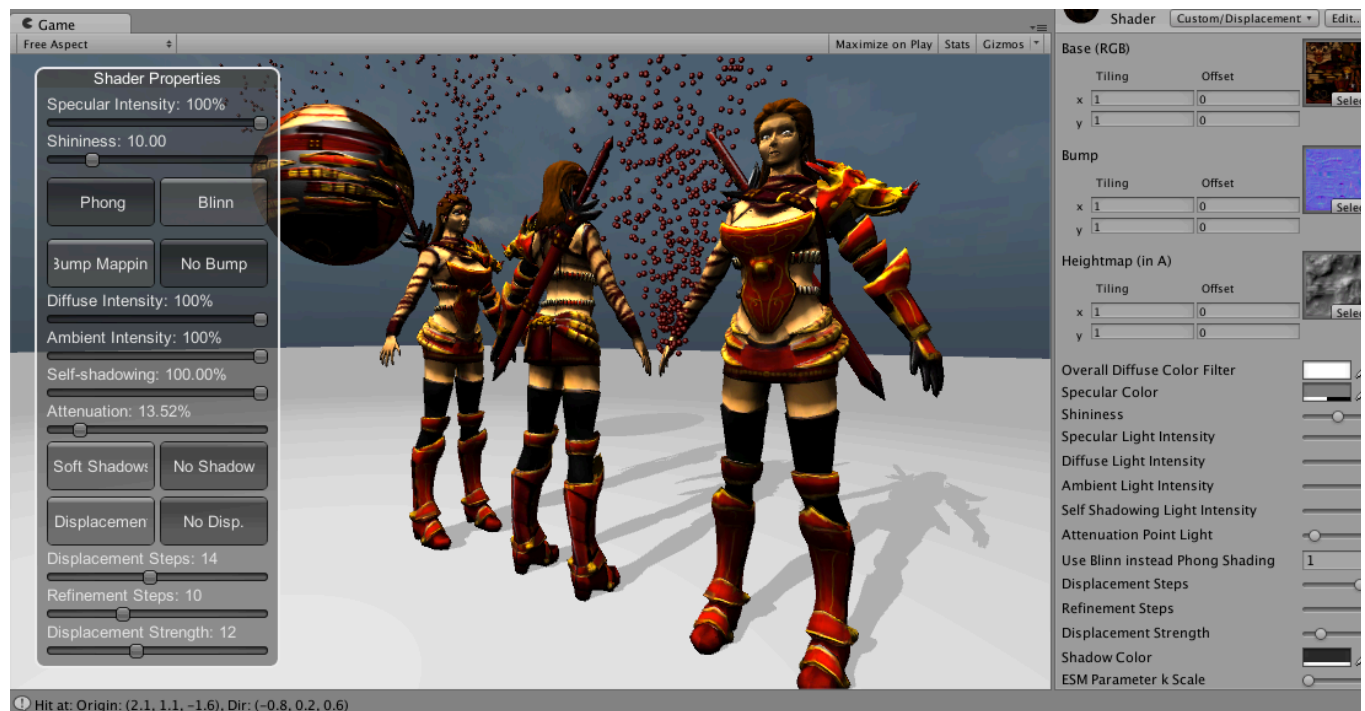


## On-The-Fly wechselbare und konfigurierbare Shader in Unity3D Free

### Beschreibung:

Dieses Projekt demonstriert den Einsatz von anpassbaren Shadern in **Unity 4.0 Free**. Die implementierten Techniken und Features wurden in der Shader-Programmiersprache **Cg** entworfen und werden auf der GPU ausgeführt. Die CPU hat dadurch freie Rechenleistung und kann andere Aufgaben übernehmen. Es werden **keine Surface-Shader** verwendet. Die Anpassung der Shadereinstellungen erfolgt mittels einer GUI, hinter welcher ein Skript via **ShaderLab Properties** die Werte des Shaders setzt. Besonderer Fokus wurde auch auf **Multiplattform**-Unterstützung gelegt. Die Anwendung wurde unter **Mac OS X** erstellt und läuft neben diesem auch auf Linux und Windows ausführbar. Die Grafikkarte muss **Shader Model 3.0** oder höher unterstützen.

### Screenshot:



### Steuerung:

Taste	Funktion
W, S, A, D	Kamerabewegung vor (W), zurück (S), links (A) bzw. rechts (D)
Rechte Maustaste + Mausbewegung	Blickrichtung verändern
Mausklick	Einen Strahl in Kamerablickrichtung schießen. Trifft Strahl auf ein Objekt, wird ein Partikelsystem an dieser Stelle erzeugt
GUI Controls	Mit den GUI Controls auf der rechten Seite können unterschiedliche Eigenschaften im Shader aktiviert oder verändert werden

## Features:

- GUI zum einstellen der Shaderproperties
- Shader (in Cg mit Vertex- und Fragment-Shaders, keine Surface-Shaders)
  - Directional und Point Light
  - dynamisch anpassbar/änderbar:
    - Blinn-Beleuchtungsmodell anstatt Phong
    - Ambiente, Diffuse und Spekuläre Intensität
    - Shininess / Glossiness spekulärer Highlights
    - Self-Shadowing Faktor des Models
    - Entfernung des Point Lights (Attenuation, mehr = dunkler)
    - Einfaches Color Mapping
    - Color Mapping + Bump Mapping
    - **Color Mapping + Bump Mapping + Displacement Mapping**
    - Schrittweite, verfeinerte Schrittweite und Stärke des Displacement Mappings
    - **Bewegte Partikel**
    - **Dynamische Schatten**
- **Strahlenschnitt mit Objekten**

## Einsetzbare, selbstentwickelte Shader:

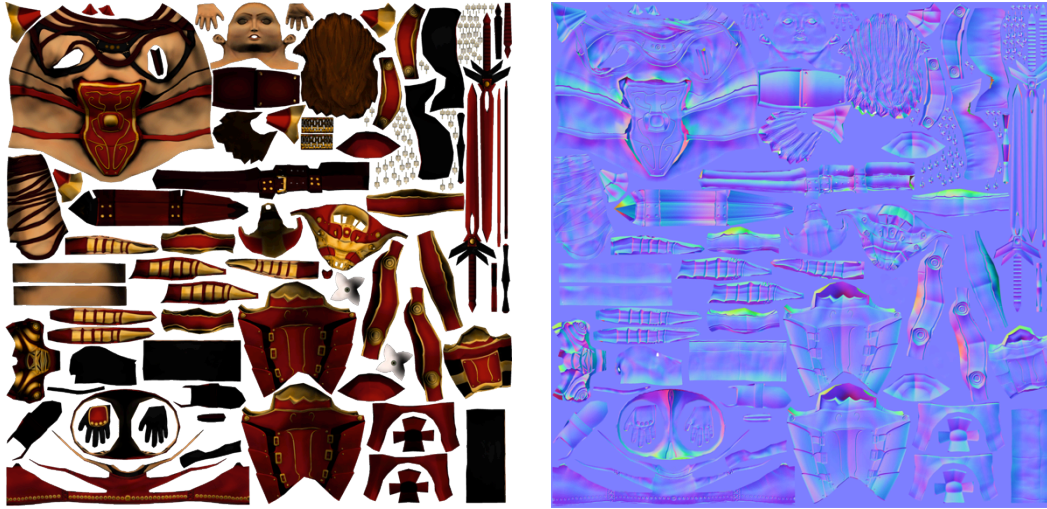
1. Default Shader (*Resources/Shaders/Default.shader*)
  - Unterstützt Direktionales- und Punkt-Licht
  - Texture-Mapping
2. Bump Shader (*Resources/Shaders/Bump.shader*)
  - Erweitert den Default Shader um Bump Mapping
3. Displacement Shader (*Resources/Shaders/Displacement.shader*)
  - Erweitert den Bump Shader um Displacement Mapping
4. Partikel Shader (*Resources/Particles/Particles.shader*)
  - Stellt sich bewegende Partikel dar
5. Shadow Shader (*Resources/Shaders/DisplacementSoftShadow.shader*)
  - Zeichnet Objektschatten auf die Ground Plane

## Techniken:

### Bump Mapping

Bump Mapping bzw. Normal Mapping dient zum Vortäuschen von geometrischen Details. Dadurch kann Objekten mit wenig Geometrie ein deutlich besseres Aussehen verliehen werden, ohne jedoch mehr Geometrie erzeugen oder rendern zu müssen. Dazu werden die Normalvektoren eines Modells anhand der Informationen in einer speziellen Textur abgeändert, was in weiterer Folge bei der Beleuchtung des Objektes eine Änderung des Farbwertes aufgrund anderer Reflexion des Lichtes bei diesem Pixel zur Folge hat. Diese spezielle Textur wird Normal Map genannt und gibt zu jedem Pixel der eigentlichen Textur eine (farbcodierte) Richtung an. Verbiegt man nun die eigentlichen Normalvektoren der Geometrie mit denen der Normal Map, entsteht eine andere Ausrichtung der Oberfläche, wodurch es wirkt, als ob die Oberfläche Unebenheiten hat und eine detailliertere Struktur aufweist. Es ist daher eine relativ günstige Art, einem Objekt ein besseres Aussehen zu verleihen. Jedoch ist das Fehlen einer echten Oberflächenstruktur recht einfach beim Betrachten der

Silhouette eines Objektes erkennbar, da dieses immer noch glatt ist und keine zusätzliche Struktur aufweist.



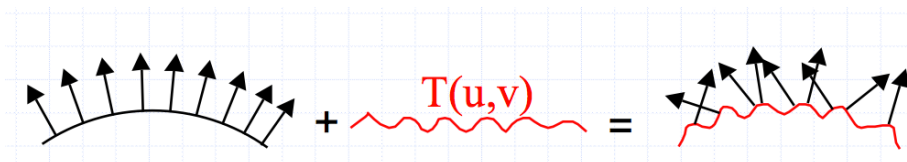
Links: Color Textur des Models, rechts: Normal Map Textur

Um diese Funktionalität zu unterstützen, wurde ein Material „bump“ angelegt, in welcher die Color Textur und die Normal Map Textur spezifiziert sind. Der mit diesem Material verknüpfte Shader ist der **Shader „BumpShader“**, welcher die neue Normalvektor Berechnung vornimmt.

### Displacement Mapping (Pixel Shader Displacement Mapping):

Bump- bzw. Normal-Mapping täuschen Details nur vor, die Silhouette bleibt jedoch unverändert. Beim Displacement Mappinging wird hingegen aus Werten einer Height-Map-Textur tatsächliche Geometrie erzeugt und das Modell damit angepasst. Mit Displacement Mapping können Vertices z.B. „hinein oder heraus“ bewegt werden, um einem Modell eine detailliertere Oberflächenstruktur zu verleihen. Das heißt, auch Modells mit weniger Geometrie können adaptiv in Echtzeit sichtlich verbessert werden.

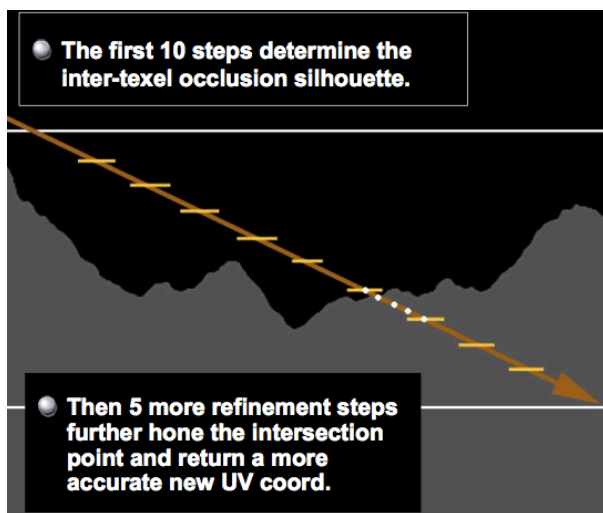
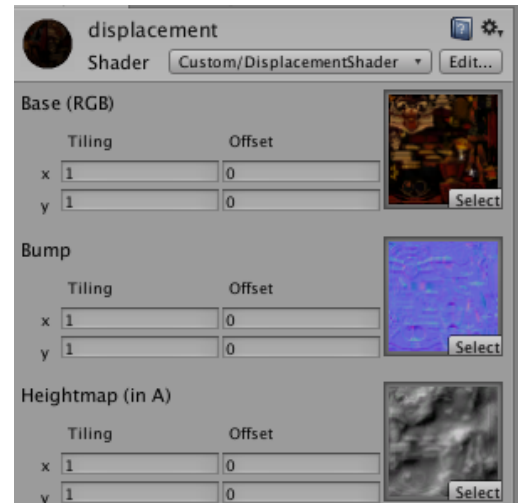
Folgende Abbildung zeigt, wie die Silhouette einer Kugel durch die Höheninformationen aus der Height-Map (in rot dargestellt) verändert wird:



Diese Abbildung zeigt das für Displacement Mapping verwendete Material „displacement“:




Im Unterschied zur Basis- und Bump-Textur fällt auf, dass die Heightmap lediglich 1 Farbkanal verwendet, da beim Displacement Mapping nur ein Höhen-Wert benötigt, um welchen der Vertex (oder Pixel) der Geometrie verschoben wird.

Als Technik für Displacement Mapping wurde das von [NVIDIA im Cascades Demo](#) verwendete **Per Pixel Shader Displacement Mapping** verwendet. Mit dieser Technik können sich Pixel auch gegenseitig verdecken. Nachfolgend wird dieser Algorithmus näher beschrieben:



Visualisierung des Displacement Mapping Algorithmus in [NVIDIA Cascades Demo Folie 93](#)

#### Algorithmus:

1. Ausgehend von der Blickrichtung wird der „eyeVector“ berechnet.
2. Für jeden Pixel marschieren wir in groben Schritten entlang dieses Vektors. Die Unterteilung des Weges (=Anzahl der Schritte) erfolgt anhand der in der GUI eingestellten Displacement Steps (Range 0 bis 30): 
3. Mit jedem Schritt wird die für den Lookup verwendete Texturkoordinate proportional zum gegangenen Weg und der eingestellten Stärke des Displacements um Delta-UV angepasst. Diese Stärke kann über den Slider Displacement Strength angepasst werden: 
4. Sobald wir auf den ersten Punkt unterhalb der Modelloberfläche treffen, merken wir uns die Schrittweite, da wir die For-Schleife nicht einfach abbrechen, sondern bis zum letzten Schritt durchwandern müssen.
5. Von dem Punkt, an dem wir das erste Mal die Oberfläche getroffen haben, gehen wir einen Schritt zurück, sodass wir wieder oberhalb der Oberfläche sind. Von dort wird ein neuer Marsch mit feineren Schritten begonnen, sodass wir möglichst Nahe an der Oberfläche auftreffen. Der zweite Marsch kann mittels der Einstellung Refinement Steps verfeinert werden: . Je mehr Refinement Steps, desto mehr Schritte werden verwendet, desto kürzer der zurückgelegte Weg mit jedem Schritt, desto schöner und genauer das Ergebnis.

6. Die Texturkoordinaten des Pixels werden nun an die des finalen Auftreffpunktes angepasst und die Höheninformation an dieser Stelle der HeightMap ausgelesen.
7. Die finale Texturkoordinate verschiebt sich in Folge anhand der delta UV-Position zum Aufschlagpunkt unter Einbeziehung der Höhe.

Anmerkung: Da der Shader sehr instruktionsreich ist und auch for-Schleifen verwendet, ist das Shader Model 3.0 oder höher erforderlich. Adressiert wurde dieses mit dem Befehl:

```
#pragma target 3.0
```

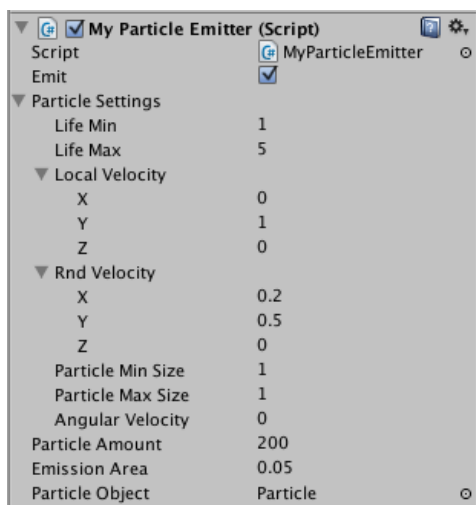
## Strahlenschnitt und Partikel:

### Strahlenschnitt:

Unity vereinfacht die Kollisionsberechnung mit Objekten enorm. Folgende Schritte waren für den Strahlenschnitt nötig:

1. Den Models wurde die mächtige Physik-Komponente Mesh Collider hinzugefügt, die automatisch Collider-Geometrie für das Mesh erzeugt.
2. Der Hauptkamera wurde ein Skript hinzugefügt, welches prüft, ob die Linke Maustaste gedrückt wurde.
3. Wurde die Maustaste gedrückt, wird mit folgendem Befehl ein Strahl an der Mausposition in Richtung der Kamera-Blickrichtung erzeugt:  
`Ray ray = camera.ScreenPointToRay (Input.mousePosition);`
4. Dieser Strahl wird in Folge mit Unity's Physik Engine in die Szene geschossen:  
`Physics.Raycast(ray, out hit)`
5. Sobald der Strahl auf einen Collider trifft, endet die Berechnung und die Position mitsamt dem getroffenen Collider wird zurückgegeben.
6. An dieser Position wird ein ParticleEmitter instanziiert:  
`Instantiate(particleEmitter, hit.point, transform.rotation);`

### Particle Emitter und Particles:



Der Partikel Emitter hält die Informationen zu den Eigenschaften wie Lebensdauer, Richtung, zufällig abweichende Richtung, Partikelgröße und Anzahl der Partikel. Es berechnet zudem die tatsächliche

Startposition der Partikel durch Zufallsgenerierung eines Punktes auf einer Sphere  
(Random.onUnitSphere) skaliert um den Faktor der Emission Area rund um die Collider-Hit-Position.

Ursprünglich wurde versucht, die Partikel im Geometry Shader zu erzeugen. Einerseits wird hierfür das Direct X **Shader Model 4.0** benötigt, andererseits funktioniert die Unterstützung des Geometry Shaders in Unity nur unter Verwendung von **DirectX 11**:

- **#pragma geometry name** - compile function *name* as DX10 geometry shader. Having this option automatically turns on **#pragma target 4.0**, see [below](#).
- **#pragma target 4.0** - compile to DX10 shader model 4.0. This target is currently only supported by DirectX 11 renderer.

Quelle: <http://docs.unity3d.com/Documentation/Components/SL-ShaderPrograms.html#target>

Die Problematik hierbei ist daher, dass die Verwendung des Geometry Shaders unter Mac OS X nicht möglich ist und die Anwendung zudem auf Windows eingrenzen würde. Die Partikelgenerierung musste daher so gelöst werden, dass im ParticleEmitter ein Particle-GameObject instanziiert wird und um eine Komponente ParticleController erweitert wird, welche das GameObject nach einer in den Settings eingestellten Lebensdauer wieder löscht. Im Partikel-Shader wird bei der Erstellung das ShaderLab Property „birthTime“, also die Geburtszeit (Time.time), als auch der Richtungsvektor gesetzt. Durch die Richtungseigenschaften des Partikels wird in Folge beim Rendering die aktuelle Position anhand der Startposition plus dem Richtungsvektor \* Lebensdauer berechnet.

## Dynamische Schatten

Der Depth Buffer, auch als Z-Buffer bekannt, dient der Speicherung der Tiefeninformation eines Pixels und eignet sich perfekt zur Verwendung beim Shadow Mapping. Bei diesem wird bei jedem Renderdurchgang zu Beginn für das Shadow Mapping eine 2D Textur gerendert, welche unter Zuhilfenahme des Depth Buffers die Tiefeninformation der Szene aus der Sicht des Lichtes in eine Shadow Map speichert. Beim eigentlichen Renderdurchgang wird nun die Tiefe des Wertes in der Shadow Map mit der Entfernung des Lichtes verglichen, um festzustellen, ob sich ein Pixel im Licht oder im Schatten befindet. Wenn sich das Licht vor dem Objekt befindet, wird dieser Pixel beleuchtet, ansonsten ist dieser verdeckt und wird nicht bzw. nur wenig beleuchtet. Diese Logik wird im Shader abgewickelt. Mittels Soft-Shadowing Techniken wie PCF (Percentage Closer Filtering) oder ESM (Exponential Shadow Maps) können beim Rendern der Shadow Map die harten Schatten mit Halbschatten erweitert werden, wodurch die Schatten weicher und realistischer erscheinen.

Da Unity den Aufruf der Rendering Passes von Objekten übernimmt, ist eine gezielte Abfolge des Renderings, bei dem zuerst alle Modelle aus Sicht des Lichts in den Depth-Buffer gerendert und der Depth-Buffer nachfolgend beim Rendern der Ground Plane zum auftragen der Schatten herangezogen wird, nicht möglich.

