Computergrafik Praktikum 2

Allgemeines zum Praktikum

Im Rahmen des Computergrafik-Praktikums lernen Sie, die aus der Vorlesung gewonnenen Erkenntnisse praktisch mit der Programmiersprache C/C++ umzusetzen.

Bei der Konzipierung der Praktika wurde darauf geachtet, dass die verwendeten Bibliotheken sowohl für Windows als auch für OS X erhältlich sind. Sofern Sie die Aufgaben im Medienlabor bearbeiten, steht Ihnen die Entwicklungsumgebung XCode zur Verfügung, mit der Sie Ihre C++-Programme entwickeln können. Wenn Sie die Aufgaben an Ihrem privaten Rechner unter Windows bearbeiten möchten, empfehle ich Ihnen die Entwicklungsumgebung Visual Studio 2013, die Sie als Student/-in über das Microsoft ELMS-Programm kostenlos herunterladen können.

Bitte beachten Sie bei der Bearbeitung der Praktikumsaufgaben, dass Ihre Lösungen für folgende Praktikumsaufgaben weiter genutzt werden müssen. Die Praktikumsarbeiten bauen also ineinander auf. Bearbeiten Sie deshalb die Lösungen sorgfältig und löschen Sie diese nicht, wenn die Aufgabe abgenommen wurde. Achten Sie bei der Entwicklung darauf, dass nur die Schnittstellen in den Header-Dateien stehen, während die Implementierung ausschließlich in den cpp-Dateien steht.

Zum Bestehen des kompletten Praktikums müssen Sie jede Praktikumsaufgabe erfolgreich abschließen und mindestens 80% der Gesamtpunkte für das komplette Praktikum erhalten. Wie viele Punkte Sie für eine einzelne Aufgabe erhalten, hängt von Ihrer Implementierung und Ihren Erläuterungen ab.

Um dieses Praktikum zu bestehen, müssen Ihre entwickelten Teillösungen lauffähig sein (5 Punkte) und Sie müssen die einzelnen Lösungen erläutern können (weitere 5 Punkte). Insgesamt können Sie bei diesem Aufgabenblatt 10 Punkte + 2 Zusatzpunkte erhalten.

Thema Praktikum 1

Ziel der ersten beiden Praktikumsaufgaben (Praktikum 1 & 2) ist es, einen einfachen Raytracer zur Offline-Bildsynthese zu programmieren. Während Sie im letzten Praktikum die drei Hilfsklassen Vector, Color & RGBImage entwickelt haben, müssen Sie jetzt die Funktionalität des Raytracing-Algorithmus implementieren. Im Wesentlichen besteht diese aus den folgenden Teilimplementierungen:

- Implementierung einer Kamera-Klasse zum Erstellen von Kamerastrahlen,
- Implementierung des Strahlenverfolgungsalgorithmus,
- Implementierung des Phong-Reflexionsmodells.

Damit die Aufgabe nicht zu umfangreich wird, soll ein Raytracer entwickelt werden, der nur Reflexionen aber keine Transmissionen (Brechung von Strahlen an transluzenten Oberflächen) abbildet.

Die nötigen Szenendaten zum Testen Ihrer Implementierung werden von einigen Hilfsklassen, die Sie im OSCA-Lernraum (cgutilities.h & cgutilities.cpp) herunterladen können, erzeugt.

Mit Hilfe der Klasse *Scene* können Sie eine einfache prozedurale Szene erzeugen:

```
class Scene
{
public:
    Scene( unsigned int SceneComplexity );
    virtual ~Scene();
    const Triangle& getTriangle(unsigned int Index) const;
    unsigned int getTriangleCount() const;
    const PointLight& getLight(unsigned int Index) const;
    unsigned int getLightCount() const;
```

Bei der Erzeugung eines Szenen-Objekts können Sie die Szenenkomplexität mit dem Parameter *SceneComplexity* steuern. Setzen Sie *SceneComplexity* auf 0, so besitzt die resultierende Szene nur sehr wenige Dreiecke. Eine Szenenkomplexität von 20 dagegen besitzt mehrere tausend Dreiecke.

TIPP: Während der Implementierung des Raytracers sollten Sie die Szenenkomplexität möglichst gering einstellen, damit Ihre Implementierung schnell Ergebnisse liefert.

Die erstellte Szene besteht dann aus Dreiecken, die Sie über die Methode *getTriangle*(..) und *getTriangleCount*() abfragen können und aus Punktlichtquellen, die Sie über *getLight*(..) und *getLightCount*() abfragen können.

getTriangle(..) liefert ein Objekt des Typs *Triangle* zurück, das die folgende Schnittstelle besitzt:

```
class Triangle
{
public:
    Triangle();
    Triangle(const Vector& a, const Vector& b, const Vector& c, const Material* mtrl);
    Vector A,B,C; // vertex-positions
    const Material* pMtrl; // pointer to triangle material
    Vector calcNormal( const Vector& PointOnTriangle) const;
```

Die Membervariablen *A*, *B* und *C* sind die Eckpunkte des Dreiecks, während *pMtrl* auf das Material des Dreiecks zeigt. Die Methode *CalcNormal*(..) liefert für einen übergebenen Punkt (der auf dem Dreieck liegen muss!) eine Normale zurück (diese variiert innerhalb des Dreiecks).
Das Material-Objekt, auf das *pMtrl* zeigt, beschreibt ein Material nach dem Phong-Reflexionsmodell:

```
class Material
{
public:
    Material();
    Material( const Color& Diffuse, const Color& Specular, const Color& Ambient, float SpecularExp,  float Reflectivity);
    virtual Color getDiffuseCoeff(const Vector& Pos) const;
    virtual Color getSpecularCoeff(const Vector& Pos) const;
    virtual Color getAmbientCoeff(const Vector& Pos) const;
    virtual float getSpecularExp(const Vector& Pos) const;
    virtual float getReflectivity(const Vector& Pos) const;
```

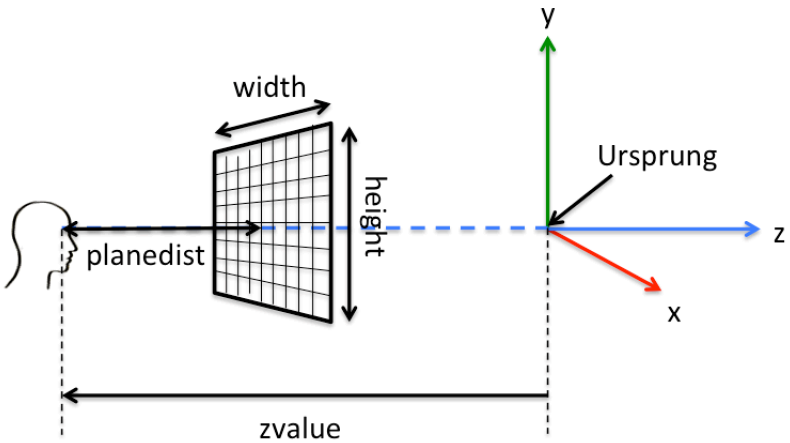
getDiffuseCoeff(..) liefert den diffusen Reflexionskoeffizienten k_d.

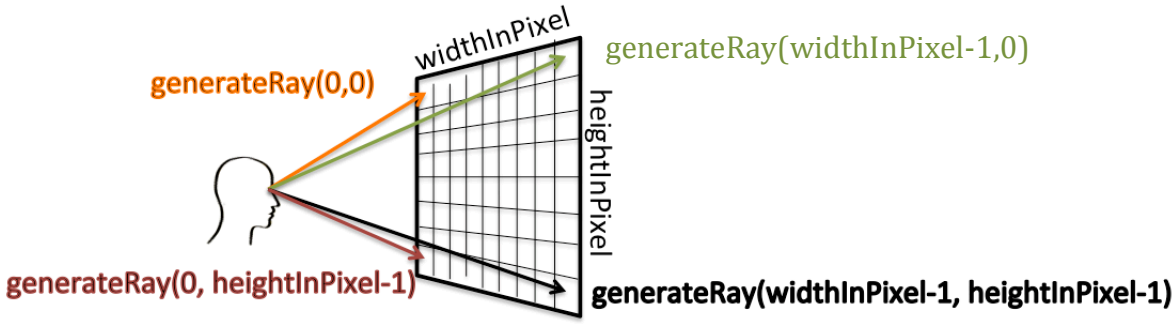
getSpecularCoeff(..) liefert den spekularen Reflexionskoeffizienten k_s.
getSpecularExp(..) liefert den spekularen Exponenten n.
getAmbientCoeff(..) liefert die ambiente Helligkeit I_a.
getReflectivity(..) liefert Ihnen den Reflexionskoeffizienten für den Spiegelstrahl. Transmissionsstrahlen müssen Sie bei dieser Aufgabe nicht berechnen, weswegen auch kein Transmissionskoeffizient vom Material abfragbar ist.

Aufgabe 1 (2 Punkte)

Implementieren Sie eine Kameraklasse, die immer entlang der positiven Z-Achse ausgerichtet ist und die folgenden Parameter aufweist:

- planedist: Abstand zur Bildebene.
- width: Breite der Bildebene in Raumeinheiten.
- height: Höhe der Bildebene in Raumeinheiten.
- zvalue: Die Position des Augpunkts auf der Z-Achse.
- widthInPixel: Die Breite der Bildebene in Pixeln (z. B. 640 Pixel)
- heightInPixel: Die Höhe der Bildebene in Pixeln (z. B. 480 Pixel)





Die Schnittstelle der *Camera*-Klasse sollte wie folgt aussehen:

```
class Camera
{
public:
    Camera( float zvalue, float planedist, float width, float height, unsigned int widthInPixel, unsigned int heightInPixel );
    Vector generateRay( unsigned int x, unsigned int y) const;
    Vector Position() const;
```

Bemerkungen:

- *Position*() liefert die Position des Augpunkts im 3D-Raum.

- *generateRay*(..) liefert für die Pixelkoordinaten x & y die Richtung eines Strahls, der vom Augpunkt durch den Pixel schießt. Der zurückgegebene Strahl muss die Länge eins besitzen (zur Verdeutlichung des Prinzips siehe Abbildung oben).
- Den internen Aufbau der Klasse können Sie selbst bestimmen.

Aufgabe 2 (3 Punkte)

Implementieren Sie den Strahlenverfolgungsalgorithmus aus der Vorlesung. Der Algorithmus soll durch die folgende Schnittstelle gestartet werden:

```
class SimpleRayTracer
{
public:
    SimpleRayTracer(unsigned int MaxDepth);
    void traceScene( const Scene& SceneModel, RGBImage& Image);

protected:
    Color trace( const Scene& SceneModel, const Vector& o, const Vector& d, int depth);
    Color localIllumination( const Vector& SurfacePoint, const Vector& Eye, const Vector& Normal, const PointLight& Light, const
        Material& Material );
    int m_MaxDepth;

};
```

Bemerkungen:

- Um die Aufgabe etwas zu vereinfachen, müssen Sie für die Strahlenverfolgung keine Transmissionen berücksichtigen.
- Die Methode traceScene wird von außerhalb der Klasse aufgerufen, um den Raytracing-Algorithmus für die übergebene Szene *SceneModel* zu starten. Das sich ergebene Bild muss dann im Image-Objekt abgelegt werden. Die Verwendung der Klasse sieht wie folgt aus:

```
int main(int argc, char **argv) {
    Scene ModelScene(0);
    RGBImage Image(640,480);
    SimpleRayTracer Raytracer(2);
    Raytracer.traceScene(ModelScene, Image);
    Image.saveToDisk("raytracing_image.bmp");
}
```

- In dieser Aufgabe ist es noch nicht erforderlich, die Methode *localIllumination*(..) korrekt zu implementieren. Stattdessen können Sie in dieser Methode einfach die Diffuse-Komponente des Materials zurückgeben:
  `return Material.getDiffuseCoeff(SurfacePoint);`
- Bei der Konstruktion der Klasse soll eine maximale Rekursionstiefe *MaxDepth* übergeben werden.
- In der Methode *traceScene* werden die initialen Kamerastrahlen erzeugt und durch den rekursiven Aufruf von *trace*(..) verfolgt. Initialisieren Sie die Kamera mit den folgenden Werten:
  zvalue=-8, *Planedist*=1, *width*=1, *height*=0.75, *widthInPixel*=640, *heightInPixel*=480.
- Sie können den internen Aufbau der Klasse selbst bestimmen, fügen Sie also ggf. weitere Hilfsmethoden etc. hinzu.

Aufgabe 3 (3 Punkte)

Implementieren Sie das Phong-Reflexionsmodell in der Methode *localIllumination*(..). Erläuterungen zur Material-Klasse finden Sie im allgemeinen Teil der Aufgabenbeschreibung. Die Parameter der Methode *localIllumination*(..) sind wie folgt definiert:

- *SurfacePoint*: Punkt auf der Oberfläche (Schnittpunkt Oberfläche mit Strahl),
- *Eye*: Position des Augpunkts,
- *Normal*: Normale des Dreiecks (übergeben Sie hier *Triangle::calcNormal*(...)),
- *Light*: Eine Punktlichtquelle (Position und Intensität der Lichtquelle),
- *Material*: das Material des Dreiecks.

Bemerkungen:
- Lassen Sie bei der Berechnung der Phong-Reflexion den Abstand der Lichtquelle unberücksichtigt (keine Helligkeitsabnahme mit der Entfernung).

Aufgabe 4 (2 Punkte)

Rendern Sie die Szene mit den folgenden Einstellungen und speichern Sie das Ergebnis unter „Prakt2aufg4.bmp“ ab:

SceneComplexity=20,
MaxDepth=2,
zvalue=-8,
Planedist=1,
width=1,
height=0.75,
widthInPixel=640,
heightInPixel=480.

Aufgabe 5 (Zusatzaufgabe, 2 Punkte)

Erweitern Sie Ihren Raytracer so, dass dieser auch Refraktionen (Transmissionen) unterstützt. Erweitern Sie hierfür die Material-Klasse um einen Brechungsindex und ergänzen Sie zwei Methoden zur Berechnung des Reflexions- und Transmissionskoeffizienten nach Schlick. Parametrisieren Sie die Materialien so, dass das innere Modell aus Glas besteht (n=1,33).