



VEKTORIA-MANUAL

ERSTE

PROGRAMME



Game Design

Inhalt



/// HALLO WELT!

/// HALLO KUGEL!

/// TEXTURIERTE KUGEL

/// VIRTUELL VS. REAL

/// VEKTORIA-NOTATION



/// PROF. DR. TOBIAS BREINER
/// VEKTORIA MANUAL

2 VON 88
/// ERSTE PROGRAMME



|||| HALLO |||| WELT!

2 / 11

3

4

5



PROF. DR. TOBIAS BREINER
VEKTORIA MANUAL

3 VON 88
ERSTE PROGRAMME

Vorgreifen

Um Sie zu motivieren, greifen wir didaktisch etwas vor, und lassen Sie erst einmal das kleinstmögliche sinnvolle Programm schreiben, welches mit Vektoria möglich ist, das „Hallo Welt“-Programm, welches nur „Hallo Welt!“ auf den Bildschirm schreibt.

Danach folgt das „Hallo Kugel!“-Programm, welches eine 3D-Kugel auf den Bildschirm zaubert.

Die darin verwendeten Befehle werden danach erklärt.

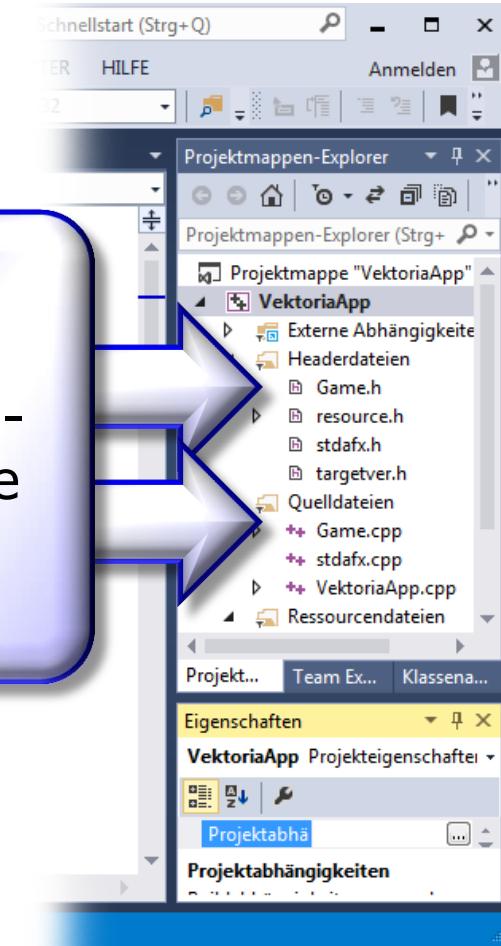


Hallo Welt-Anwendung

Wir arbeiten in Game-Klasse

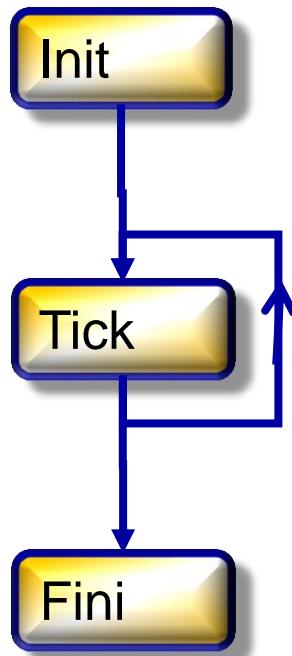
Um ein neues Vektoria-Projekt zu erstellen, kopieren Sie einfach den Template-Ordner „App“, benennen den kopierten Ordner sinnfällig um, öffnen das darin befindliche Projekt und programmieren Innerhalb der Game-Klasse.

Dort finden Sie die Methoden **Init**, **Tick** und **Fini**. Außerdem die Methode **WindowReSize**.



Hallo Welt-Anwendung

Die "heilige Dreifaltigkeit" eines Spiels



- { Initialisierung aller Variablen,
Aufbau der Objekthierarchie
- { Interaktive Eingabe von
Benutzer &
Diskreter Berechnungsschritt
eines Zeitdeltas
- { Ausgabe des Gewinners &
Finalisierung





Knotenobjekte von Szenegrafen (allgemein)

Root

Funktion:

- Ist Wurzel und Eintrittspunkt des Szenengrafen.
- An ihm hängen alle anderen Knotenobjekte

Synonyme in anderen Szenegrafen:

- Origin, Overgroup, Base

Zu beachten ist:

- Root ist keine Basisklasse!
- Es kann nur eine Wurzel geben => oft als Singleton realisiert (in Vektoria nicht)



CRoot-Grundmethoden



1 // / / / void Init(CSplash * psplash);

Initialisiert die Wurzel.
Braucht einen Zeiger
zum Startbild
(splash screen).

2 // / / / void Tick(float & fTimeDelta);

Diese Methode muss
jedes Frame aufgerufen
werden. Der Parameter
fTimeDelta gibt die Zeit
in Sekunden seit dem
letzten Tick an.

3 // / / / void Fini();

Finalisiert die Wurzel
(optional).



CRoot-Verknüpfungsmethoden

1 // / / / void AddScene(CScene * pscene);

Hängt eine Szene an die Wurzel an

2 // / / / void AddFrameHere(CFrame * pframe);

3 // / / / Hängt an den aktuellen Computer einen Frame an.
Einschränkung: Es können bis maximal vier Frames an eine Root angehangen werden
(mehr Frame-Fenster/Computer machen aber sowieso keinen Sinn, man sollte dafür lieber Viewports verwenden)

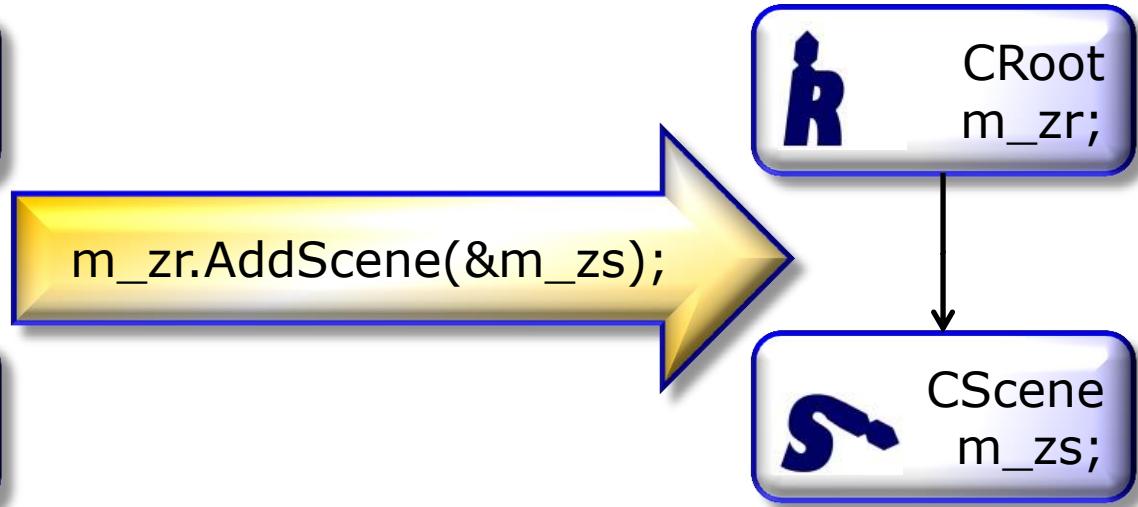
4 // / / /



CRoot - Verknüpfungsmethoden



- 1 // / / /
- 2 // / / /
- 3 // / / /
- 4 // / / /
- 5 // / / /



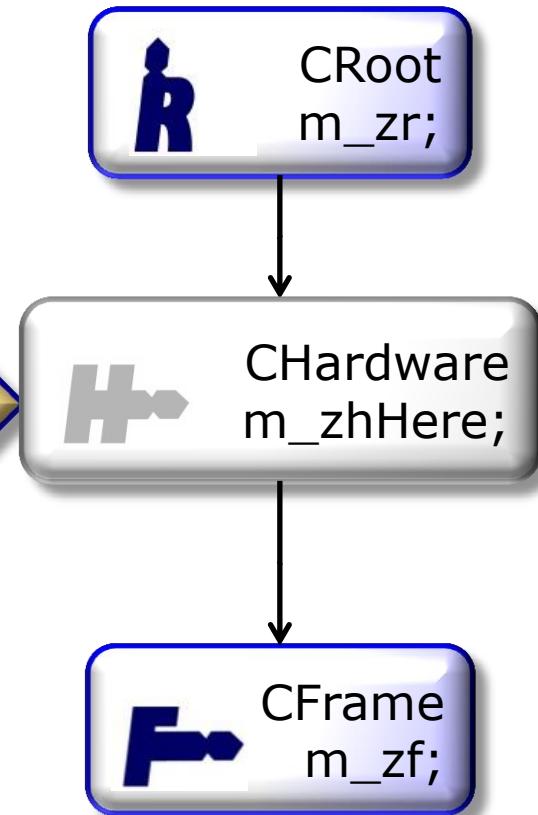


CRoot - Verknüpfungsmethoden

- 1 // / / /
- 2 // / / /
- 3 // / / /
- 4 // / / /
- 5 // / / /



m_zr.AddFrameHere
(&m_zf);





Frame

Ein Frame ist ein rechteckiger Renderbereich, welcher mit einem Fensterrahmen (Window) verknüpft ist.

Synonyme in anderen Szenografen:

Channel, Renderchannel, Render-API, Port

Mögliche Parameter:

- Pointer zum Window-Rahmen, in dem der Frame liegt.
- horizontale und vertikale Auflösung.
- Konkatenation mit einer Basis-Graphik-API (OpenGL, DirectX11, Nullrenderer ...).
- Konkatenation mit einer Basis-Input-API (Null, **Mögliche weitere Parameter in anderen Szenografen:**
- 2D-Position auf dem Sichtsystem (x_s, y_s).



Knotenobjekte der Szenegraphen

Sichtkanäle



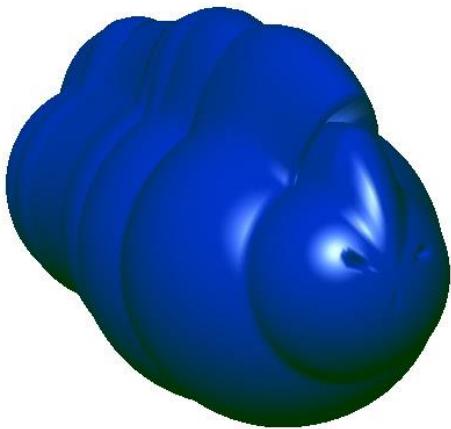
1 // / / /

2 // / / /

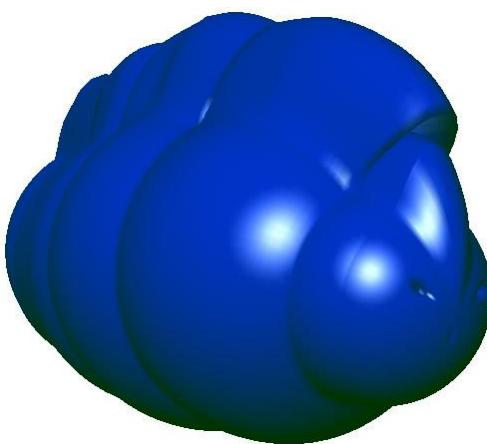
3 // / / /

4 // / / /

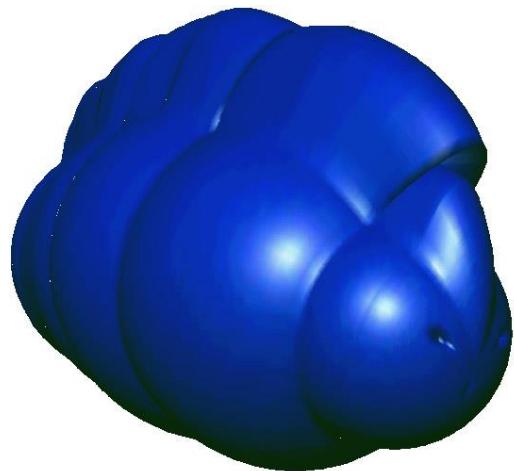
5 // / / /



Open GL



Direct 3D



Renderware16



CFrame::Init

```
CRenderApi* Init(HWND hwnd,
    EApiRender eApiRender
        = eApiRender_DirectX11_Shadermode150,
    EApiInput eApiInput
        = eApiInput_DirectInput,
    EApiSound eApiSound
        = eApiSound_DirectSound,
    EShaderCreation eShaderCreation
        = eShaderCreation_CompileChanges,
    EShaderAutoRecompilation eShaderAutoRecompilation
        = eShaderAutoRecompilation_Enabled);
```

Initialisiert den Frame und gibt einen Zeiger auf die automatisch kreierte RenderApi zurück. Der Methode muss als ersten Parameter zwingend ein ein Handle auf das Windowsfenster, in das CFrame hineinmalt, übergeben werden (**hwnd**). Dazu existieren fünf optionale Parameter, die auf den nächsten Seiten detailliert erklärt werden:
eApiRender, **eApiInput**, **eApiSound**,
eShaderCreation und **eShaderAutoRecompilation**.

CFrame::Init (eApiRender)



eApiRender bestimmt die gewünschte Grafik-Basis-Api.

Zurzeit sind folgende Eingabewerte möglich:

- **eApiRender_DirectX11_Shadermodel41**
Abgespeckter DirectX11-Renderer mit Shader Modell 4.1.
Diese Option ist sinnvoll, wenn Sie einen älteren Rechner
(um 2009) besitzen, der nicht Shadermodell 5.0 unterstützt.
- **eApiRender_DirectX11_Shadermodel50**
DirectX11-Renderer mit Shader Modell 5.0.
Dies ist die Default-Einstellung.
- **eApiRender_DirectX11_ForwardPlus**
Bester Renderer mit ForwardPlus-Technologie.
Basiert ebenfalls auf DirectX11 mit dem Shader Modell 5.0.
Allerdings ist die Parameterjustierung bei Lichtern komplex.
- **eApiRender_Null**
Null-Renderer, bei dem die Renderanweisungen ins Leere
laufen. Dies ist zum Optimieren bzw. Debuggen sinnvoll.



CFrame::Init (eApiSound)



eApiSound bestimmt die gewünschte Grafik-Basis-Api.

Zurzeit sind folgende Eingabewerte möglich:

- **eApiSound_DirectSound**
Es wird für Musik und Klänge und Geräusche die DirectSound-Api geladen (Default).
- **eApiSound_DirectAudio**
Es wird für Musik und Klänge und Geräusche die DirectAudio-Api geladen.
- **eApiSound_OpenAL**
Es wird für Musik und Klänge und Geräusche die OpenAL-Api geladen.
- **eApiSound_Null**
Null-Sound-Api, bei dem die Soundanweisungen ins Leere laufen. Kann für das Debuggen sinnvoll sein.



CFrame::Init-Parameter



eApilInput gibt die gewünschte Eingabe-API an.

Zurzeit sind zurzeit folgende Optionen erlaubt:

- **eApilInput_DirectInput (Default)**
Die Eingabe-API basiert auf DirectInput.
- **eApilInput_Null**
Die Eingabe-Abfragen laufen hier ins Leere.
Kann zum Debuggen sinnvoll sein.



CFrame::Init-Parameter



eShaderCreation gibt die Art der Shadererzeugung an.

Es sind drei Optionen möglich:

- **eShaderCreation_ForceCompile**

Hier werden alle Shader zwingend am Anfang neu kompiliert. Damit startet das Programm aber wesentlich später.

- **eShaderCreation_CompilerChanges**

Hier werden nur diejenigen Shader kompiliert, die sich seit dem letzten Mal geändert haben. Dies ist die Default-Option

- **eShaderCreation_UseCached**

Hier wird kein Shader übersetzt, sondern nur die schon übersetzten Shader aus dem Cache verwendet.

Veränderungen am Shader-Source-Code wirken sich daher nicht auf die Applikation aus.

CFrame::Init-Parameter

eShaderAutoRecompilation gibt den Zeitpunkt der Shader-Kompilierung an.

Wirkt sich naturgemäß nicht aus, falls der vorherige Parameter **eShaderCreation** auf **eShaderCreation_UseCached** gesetzt wurde.

1 Es sind folgende Optionen möglich:

- **eShaderAutoRecompilation_Disabled**

Es werden die Shader nur beim Programmstart kompiliert.

- **eShaderAutoRecompilation_Enabled**

Hier werden Shader automatisch rekompiliert, wenn sie verändert wurden, auch während der Laufzeit. Dies ist der Default-Wert.

5





Knotenobjekte von Szenographen (allgemein)

Viewport (Sichtfenster)

Viewports sind Untersichtfenster innerhalb eines Frames, welche die Sicht aus einem Kamerastandpunkt rendern.

Ein Viewport ist daher immer mit einer Kamera verbunden.

Synonyme in anderen Szenografen:

Views, Renderobject, Canvas, Port

Parameter in Vektoria:

- 2D-Position innerhalb des Frames
- 2D-Größe relativ des Frames

Zusätzliche Parameter in anderen Szenografen:

- Monoskopisch, stereoskopisch versus polyskopisch





Knotenobjekte der Szenenraphen

Viewport (Sichtfenster)

Viewports haben viele Methoden um verschiedene Einstellungen durchzuführen:

Welche grundsätzliche **Renderart** soll gewählt werden?

Drahtgittermodellanzeige (Wireframe), Polygone ohne Schattierungsinterpolation (Flat-Shading), Polygone mit Normaleninterpolation (Phong-Shading), ...

In welchem **Stil** soll der Viewport gerendert werden?

Normal, Sepia, Cartoon, Popart, ...

Welche **Features** sollen unterdrückt werden?

Schattenberechnung, Backface-Culling, Antialiasing ...

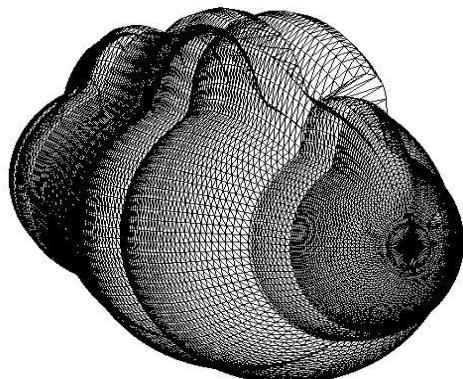
Die diesbezüglichen Methoden werden später detailliert gezeigt



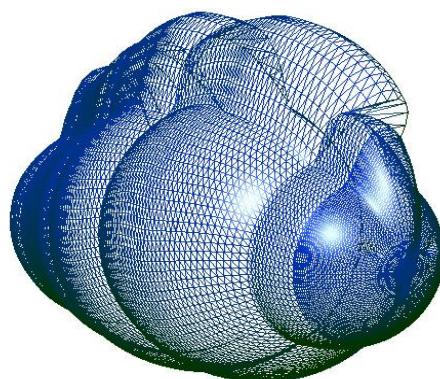


Knotenobjekte der Szenegraphen

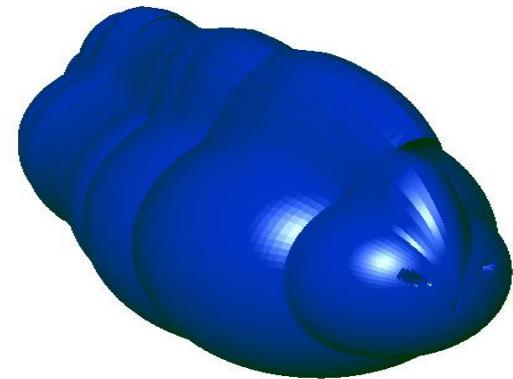
Viewport (Sichtfenster)



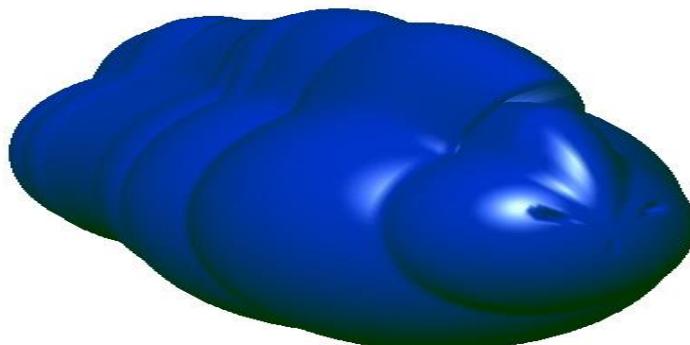
1 // Wireframe



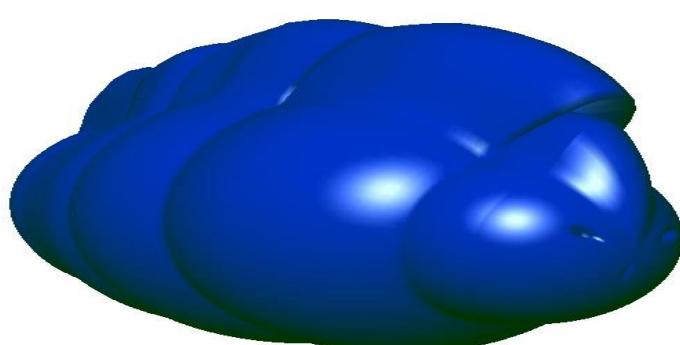
2 // Phong Shaded Wireframe



3 // Flat Shaded



4 // Phong Shaded



5 // Phong Shaded & Antialiasing



CViewport::Init

void Init(CCamera * pcamera,
float frx, float fry, float frWidth, float frHeight);

1 // / / / Initialisiert den Viewport.

2 // / / / Der Parameter **pcamera** ist ein Zeiger auf die Kamera,
welche das Sichtfenster mit Bildern versorgt.

3 // / / / Das Viewport wird mit relativen Maßen zum darüber
liegenden Frame positioniert und Skaliert.

4 // / / / oder alternativ:

5 // / / / void Init(CCamera * pcamera, CFloatRect floatrect);



CViewport::InitFull



1 // / / /

In den allermeisten Fällen benötigt man ein Viewport, welches den ganzen Frame ausfüllt, daher gibt es dafür eine Extra-Initialisierungsmethode:

2 // / / /

void InitFull(CCamera * pcamera);

3 // / / /

4 // / / /

Initialisiert den Viewport auf die volle Größe des Frames, an den er angehängt wurde.

5 // / / /





Cameras

Cameras sind die „Augen“ einer Virtuellen Szene.
Ihre Sicht kann von einem oder mehreren Viewports
angezeigt werden.

Synonyme in anderen Szenegrafen:

Eyes, Viewpoints

Parameter in Vektoria:

- Parameter: Sichtwinkel, Near Clipping Plane, Far Clipping Plane, Projektionsart

Parameter in anderen Szenegrafen:

- Typen: zielungebundene vs. zielgerichtet
- oft eigene Orientierung (Euler-Winkel),...



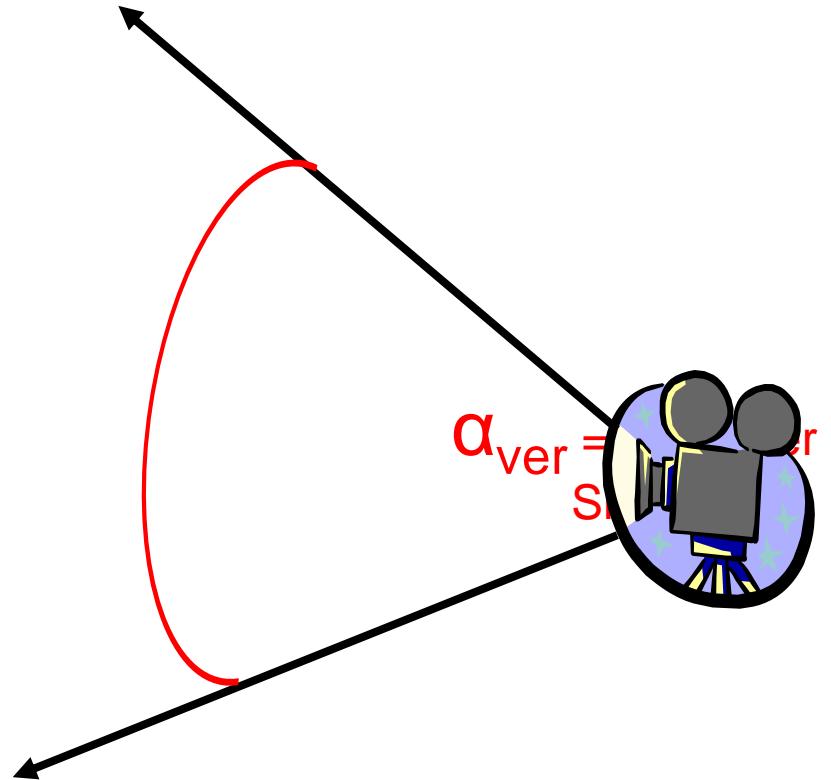
Knotenobjekte der Szenegraphen

Kameras



Es kann zwischen
horizontalem Sichtwinkel α_{hor}
und vertikalem Sichtwinkel
 α_{ver} unterschieden werden.

In Vektoria wird der vertikale
Sichtwinkel indirekt aus dem
Verhältnis zwischen Höhe h
und Breite b der Viewport-
Pixelauflösung ermittelt:



$$\alpha_{\text{ver}} = \alpha_{\text{hor}} \cdot \frac{h}{b}$$



Knotenobjekte der Szenengraphen

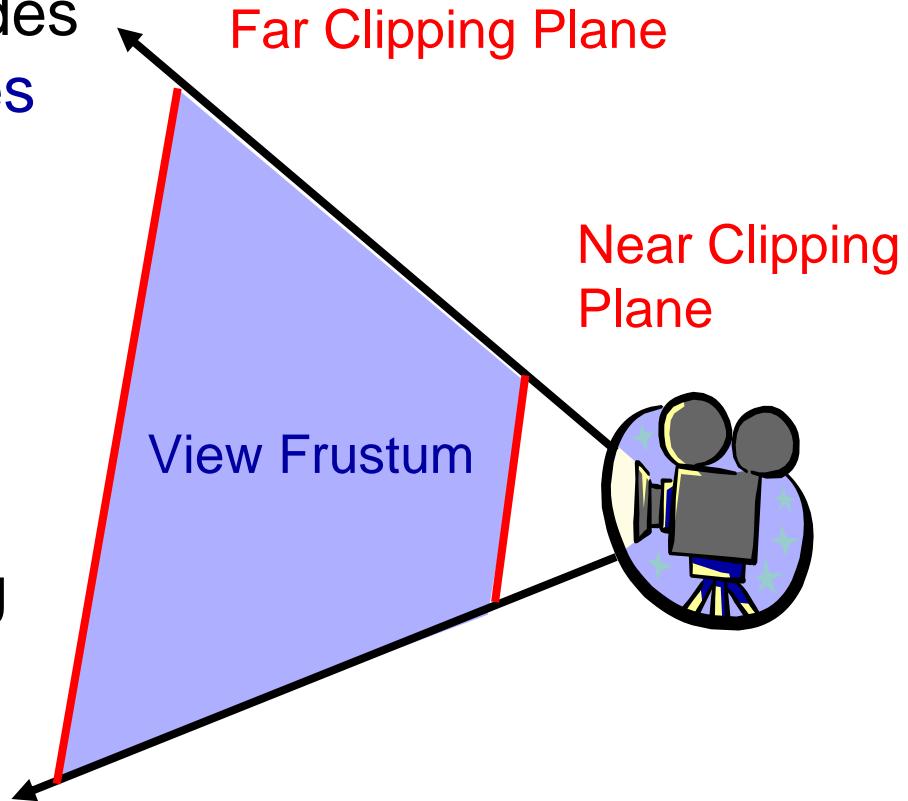
Kameras



- 1 // / / /
- 2 // / / /
- 3 // / / /
- 4 // / / /
- 5 // / / /

Nur Objekte innerhalb des Sichtpyramidenstumpfes (view frustums) werden angezeigt.

⇒ Sensible Wahl der Vorder- und Hinterschnittebene (far und front clipping plane) erforderlich!



CCamera::Init()

```
void Init( float faFovHorizontal=2.0F,  
           float fNearClipping=0.1F, float fFarClipping=1000.0F);
```

Initialisiert die Kamera.

Der Parameter **faFov** ist ein Winkel im Bogenmaß, der den Kameraöffnungswinkel in X-Richtung angibt.

Der Kamerawinkel in Y-Richtung wird daraus automatisch mitausgerechnet und braucht nicht extra angegeben zu werden.

Der Default-Wert für faFov ist 2,0 (Fischaugenwinkel), damit man mit einer hohen Wahrscheinlichkeit eventuelle Objekte in der Szene sieht.

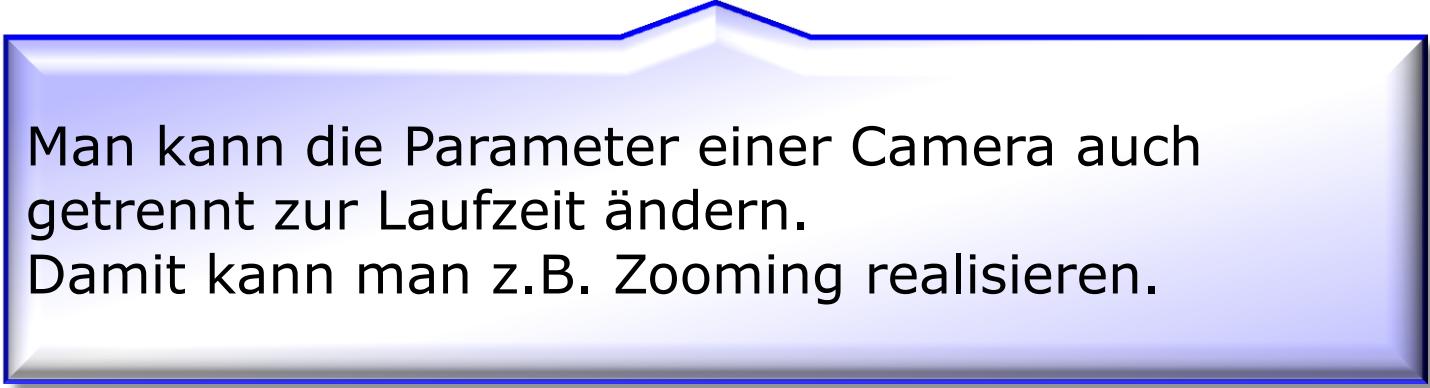
fNearClipping und **fFarClipping** geben die gewünschten Abstände zur Vorder- und Hinterschnittebene des Kamerafrustums an. Die diesbezüglichen Default-Werte sind 0,1 und 1000,0.



Weitere Methoden von CCamera



```
void SetFov(float faMat);  
void SetNearClipping Plane(float faMat);  
void SetFarClippingPlane(float faMat);
```



Man kann die Parameter einer Camera auch getrennt zur Laufzeit ändern.
Damit kann man z.B. Zooming realisieren.





Knotenobjekte der Szenegraphen

Overlay

Overlays sind 2D-Bilder, die an einen Viewport angehangen werden und stets auf seiner Oberfläche sichtbar sind.

Synonyme in anderen Szenegrafen:

Frontsprite

Parameter in Vektoria:

- Pointer zu einem Pixelbild
- Die Position des Overlays (wird in fraktionalen Werten stets relativ zum übergeordneten Viewport angegeben)
- Ein Farbschlüsselschalter (das linke, obere Pixel wird bei true als Chromakey verwendet.)





Overlay::Init



```
void Init(CImage * pimage,  
          CFloatRect & floatrect,  
          bool bChromaKeying = false);
```

1 / / / / Initialisiert das Overlay.

pimage ist ein Pointer zu einem Image.

Im Parameter **floatrect** sind die rechteckigen Ausmaße des Overlays bezüglich des übergeordneten Viewports verzeichnet.

Wenn **bChromaKeying** auf true gesetzt wird, wird das linke obere Pixel als chromatischer Schlüssel genommen, so dass alle Bildbereiche mit diesem Farbwert transparent gezeichnet werden.





Knotenobjekte von Vektoria

Image



Ein PixImage ist eine Bilddatei mit Zusatzinformationen.



```
void Init(char * acPath);
```



Initialisiert das Image durch Angabe des Pfades `acPath` der Bilddatei.



Es sind dabei absolute oder relative Pfade möglich.



Das Suffix muss bei der Pfadangabe mit angegeben werden (z.B. “`MeinBild.jpg`”)!





Overlay::Init (Schnellmethode)



```
void Init(char * acPath,  
          CFloatRect & floatrect,  
          bool bChromaKeying = false);
```



Weil ein Overlay so oft zusammen mit einem Image benötigt wird, gibt es auch eine direkte, schnelle Initialisierungsmethode



Diese Funktion initialisiert ein Overlay und ein Image anhand eines Pfades und hängt das Image direkt an das Overlay an.



4

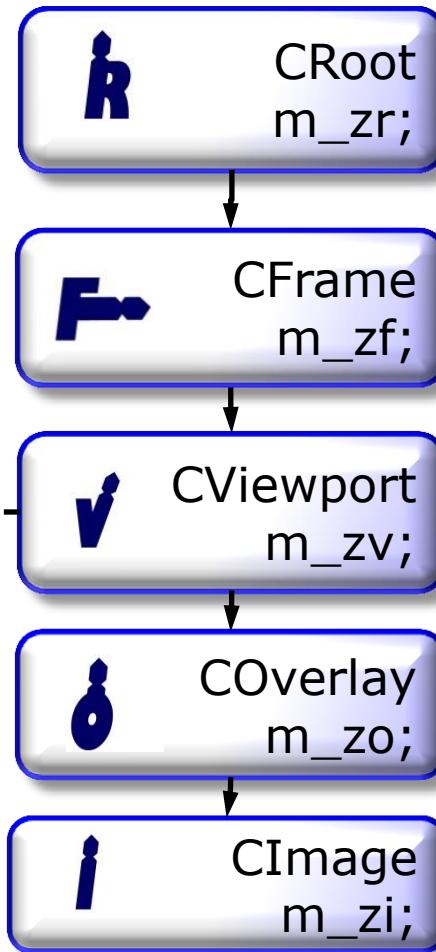


5



Hallo Welt-Anwendung

Objekthierarchie von Hallo Welt!



Objekthierarchie von Hallo Welt!

Jeder Viewport ist automatisch mit einer Kamera verbunden.



CRoot
m_zr;

Das Wurzelobjekt der gesamten Objekthierarchie. Es kann nur eine Wurzel geben.



CFrame
m_zf;

Der Renderfenster-Rahmen (in diesem Fall für DirectX)



CCamera
m_zc;



CViewport
m_zv;

In jedem Rahmen können mehrere Bilder, die Viewports, gerendert werden



COVERLAY
m_zo;

Ein Overlay ist ein 2D-Bildsprite auf einem Viewport, welches stets sichtbar ist.



CImage
m_zi;

Jedes Bildsprite braucht ein Pixel-Imagefile





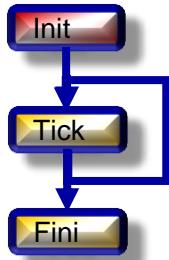
Hello Welt-Anwendung

Vektoria-Objekte in CGame.h (1/3)



```
CRoot m_zr;
CFrame m_zf;
CViewport m_zv;
CCamera m_zc;
COverlay m_zo;
CImage m_zi;
```





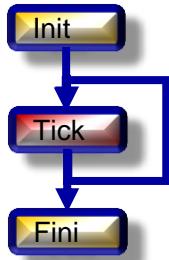
Hallo Welt-Anwendung

Init-Methode in CGame.cpp (2/3)

```
void CGame::Init(HWND hwnd, Csplash * psplash)
{
    m_zr.Init(psplash);
    m_zf.Init(hwnd);
    m_zv.InitFull(&m_zc);
    m_zo.InitFull(&m_zi);
    m_zi.Init("textures\\Hallowelt.png");

    m_zr.AddFrameHere(&m_zf);
    m_zf.Addviewport(&m_zv);
    m_zv.AddOverlay(&m_zo);
}
```





Hallo Welt-Anwendung

Tick-Methode in CGame.cpp (1/3)

```
void CGame::Tick(float fTimeDelta)
{
    m_zr.Tick(fTimeDelta);
}
```



Hallo Welt-Anwendung

Ausgabe des Hallo Welt-Programms



- 1 // / / /
- 2 // / / /
- 3 // / / /
- 4 // / / /
- 5 // / / /



Kapitel 2

Kapitel 2



/// HALLO /// KUGEL



void AddPlacement(CPlacement * pplacement);

1 //
Hängt ein Placement an eine Szene an.
Es können beliebig viele Placements an eine Szene
angehangen werden.

2 //
void AddParallelLight(CParallelLight * pparallelight);

3 //
4 //
5 //
Hängt ein Parallellicht an eine Szene an, welches dann
die ganze Szene durchflutet. Es können beliebig viele
Parallellichter an eine Szene angehangen werden.
Allerdings machen in der Praxis mehrere Parallellichter
wenig Sinn.



Knotenobjekte der Szenegraphen

Lichter allgemein



Synonyme:

Lights, Illuminations

Parameter in Vektoria:

- Lichtart (Ambient-, Spot-, Punkt, ...)
- Farbe
- Intensität (mit und ohne Entfernungsabnahme)
- Richtung & Kegelöffnungswinkel (bei Spot-Lichern)



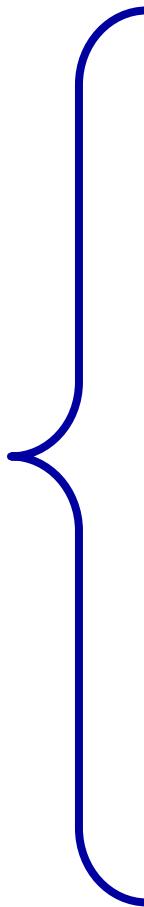
Knotenobjekte der Szenegraphen

Lichtarten allgemein



- 1 // / / /
- 2 // / / /
- 3 // / / /
- 4 // / / /
- 5 // / / /

Lichtarten



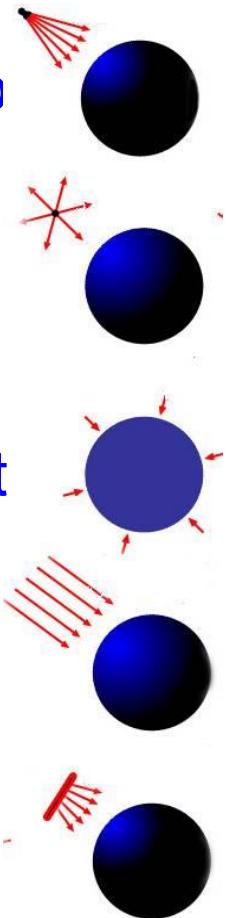
Punktlicht, **point light**

Scheinwerfer, **spo**

Hintergrundlicht, **ambient light**

Parallellicht, **parallel light**

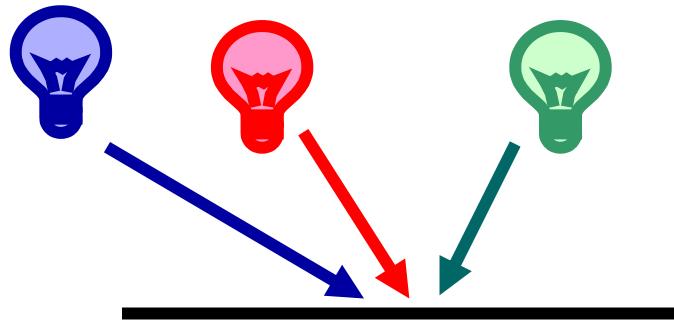
Flächenlicht, **areal light**





Lichtfarbe

- Farbige Lichter werden normalerweise im RGB-System angegeben.
- Schwache Lichtstärken werden durch niedrige RGB-Werte simuliert.
- Einige Szenographen bieten negatives Licht an (dadurch werden einfache Schatten durch negative Spotlights möglich)



CLight: Die Lichterklasse in Vektoria



CLight

CParallelLight

Parallellicht, wie z.B.
Sonnen- oder Mondlicht.

Wird an eine Szene
angehangen. Durchflutet
dann die gesamte Szene
mit parallelen Strahlen.

CPointLight

Punktlicht, wie
z.B. Glühbirnen.

Wird an ein Place-
ment angehang-
en, welches das
Zentrum der Licht-
quelle positioniert.

CSpotLight

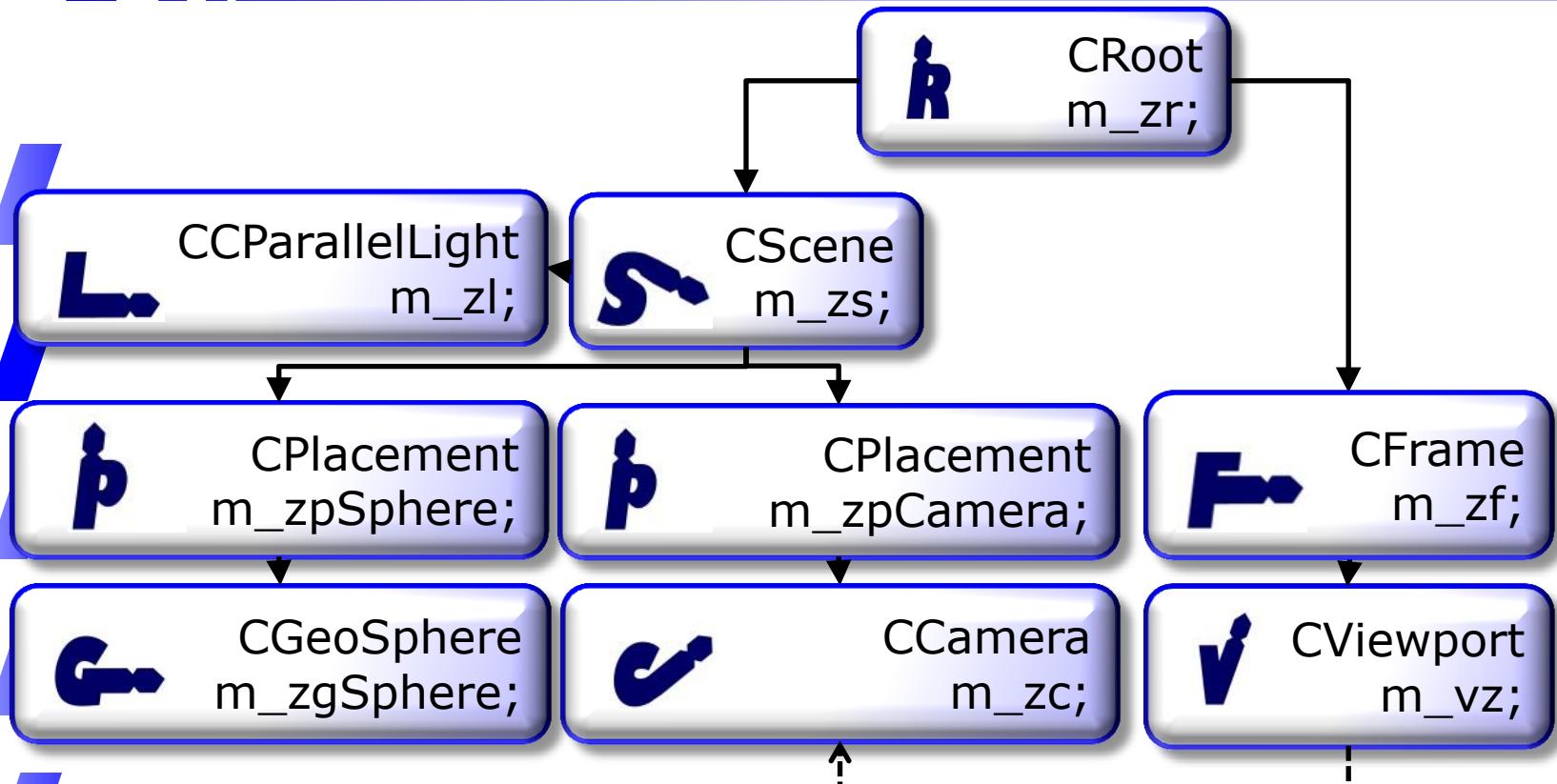
Scheinwerfer,
Taschenlampen
oder Flutlicht.

Wird an ein Place-
ment angehang-
en, welches die
Position und die
Strahlrichtung
bestimmt.



Hallo Kugel-Anwendung

Objekthierarchie von Hallo Kugel!





Hello Kugel-Anwendung

Vektoria-Objekte in CGame.h (1/5)



1 // / / /

2 // / / /

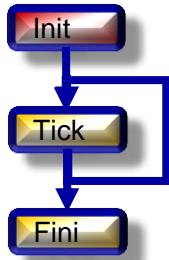
3 // / / /

4 // / / /

5 // / / /

```
CRoot m_zr;
CScene m_zs;
CPlacement m_zpCamera;
CPlacement m_zpSphere;
CGeoSphere m_zgSphere;
CFrame m_zf;
CViewport m_zv;
CCamera m_zc;
CParallelLight m_zl;
```



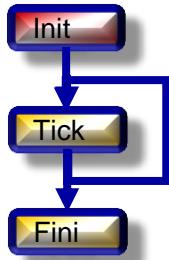


Hallo Kugel-Anwendung

Init-Methode in CGame.cpp (2/5)

```
void CGame::Init(HWND hwnd, CSplash * psplash)
{
    m_zr.Init(psplash);
    m_zc.Init(QUARTERPI);
    m_zf.Init(hwnd);
    m_zv.InitFull(&m_zc);
    m_zl.Init(CHVector(1.0f, 1.0f, 1.0f),
               CColor(1.0f, 1.0f, 1.0f));
    m_zgSphere.Init(1.5f, NULL, 50, 50);
```



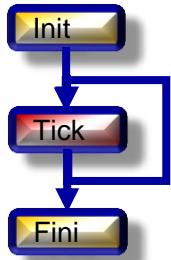


Hallo Kugel-Anwendung

Init-Methode in CGame.cpp (3/5)

```
m_zr.AddFrameHere(&m_zf);  
m_zf.Addviewport(&m_zv);  
m_zr.AddScene(&m_zs);  
m_zs.AddPlacement(&m_zpSphere);  
m_zs.AddPlacement(&m_zpCamera);  
m_zs.AddParallelLight(&m_zl);  
m_zpCamera.AddCamera(&m_zc);  
m_zpSphere.AddGeo(&m_zgSphere);  
  
m_zpCamera.TranslateZ(8.0f);  
}
```



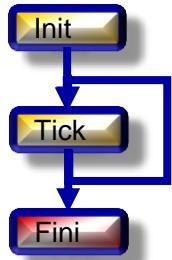


Hallo Kugel-Anwendung

Tick-Methode in CGame.cpp (4/5)

```
void CGame::Tick( float fTime,
                  float fTimeDelta)
{
    m_zr.Tick(fTimeDelta);
}
```





Hallo Kugel-Anwendung

Fini-Methode in CGame.cpp (5/5)

Die Fini-Methode braucht noch nicht befüllt zu werden.

Wir haben ja weder dynamischen Speicherplatz allokiert, der wieder freigegeben werden müsste, noch müssen wir einen Gewinner ausgeben

3 // / /

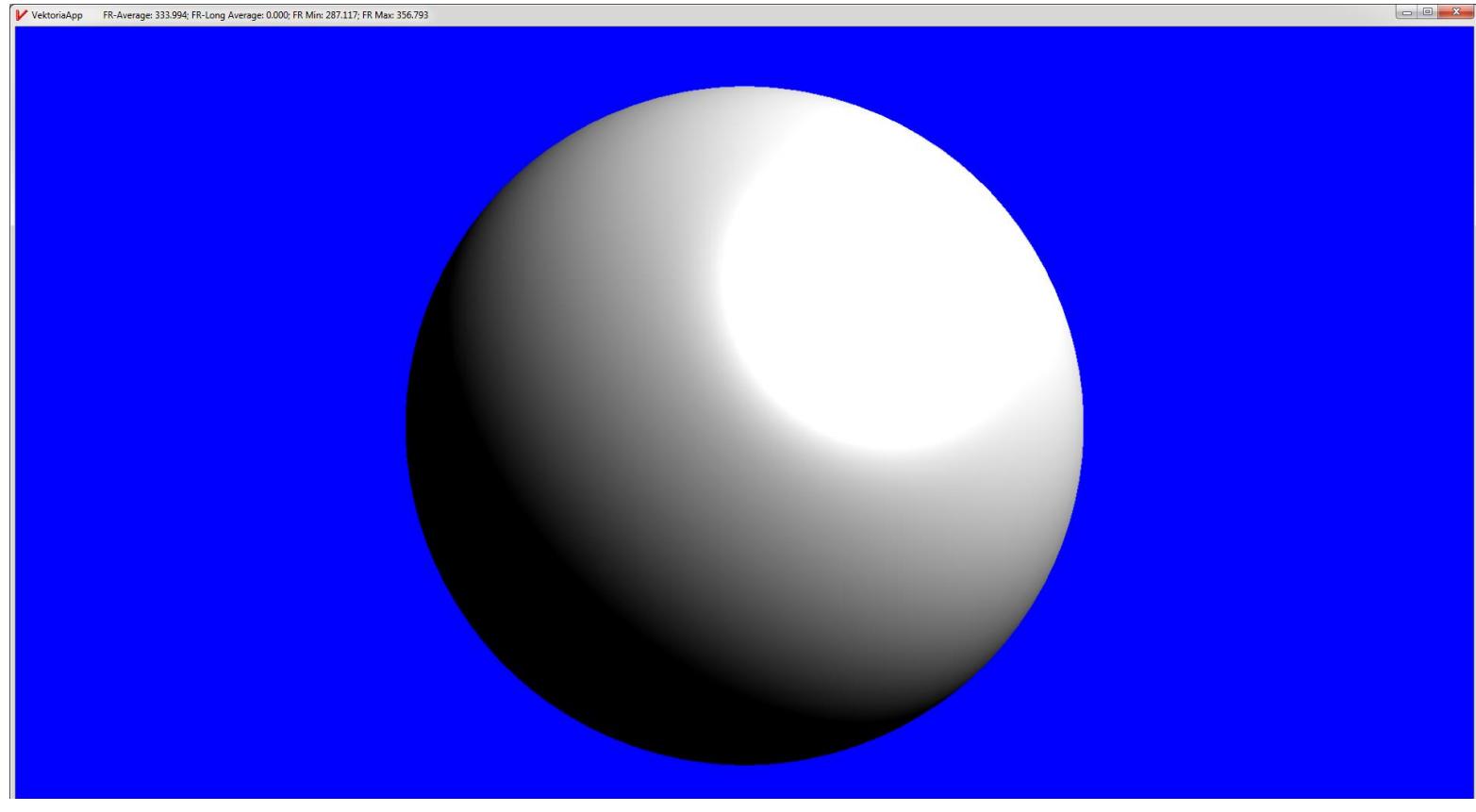
4 // / /

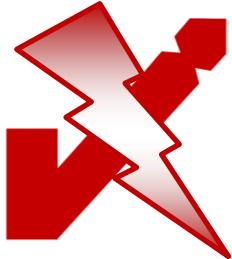
5 // / /



Hallo Kugel-Anwendung

Ausgabe





Hallo Kugel-Anwendung

Error?

1 // / / / Falls das „Hallo Kugel!“-Programm beim Ausführen abstürzen sollte, liegt es vermutlich daran, dass die Grafikkarte veraltet ist, denn Vektoria benötigt eine Shadermodell-5.0-fähige Grafikkarte (die meisten Grafikkarten nach 2010 sind 5.0-fähig).

2 // / / / Du kannst dann noch einmal versuchen, das Shadermodell auf 4.1 downzugraden (Grafikkarten um 2009), ersetze dafür :

m_zf.Init(hwnd,rectWnd.right, rectWnd.bottom,
eRenderApi_DirectX11_Shadermodel50);

3 // / / / mit:

m_zf.Init(hwnd,rectWnd.right, rectWnd.bottom,
eRenderApi_DirectX11_Shadermodel41);

4 // / / / Allerdings werden dann einige Features (Partikel, Feder-Masse-Dämpfungs-Modelle, Physik) nicht funktionieren.

5 // / / / Wenn es dann immer noch nicht gehen sollte, hilft nur noch der Kauf einer neuen Grafikkarte bzw. eines neuen Rechners.



Kapitel 3

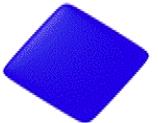
Kapitel 3



1 // / / /

2 // / / /

/// TEXTURIERTE /// KUGEL



4 // / / /

5 // / / /



PROF. DR. TOBIAS BREINER
VEKTORIA MANUAL

54 VON 88
ERSTE PROGRAMME



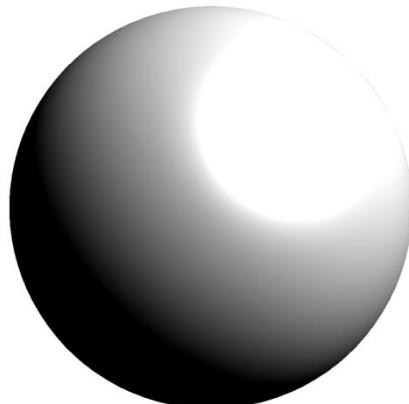
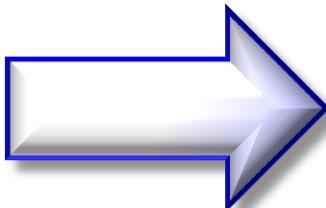
Nächste Aufgabe

Hallo Kugel mit Textur

Als nächste Aufgabe werden wir auf unsere Kugel eine Textur aufbringen. Das Aufbringen einer Textur auf eine 3D-Oberfläche nennt man „**mappen**“.

Man kann sich mappen metaphorisch vorstellen, als würde man das Objekt mit einem unendlich elastischen Gummi tapezieren.
In Vektoria muss eine Textur immer über ein „Material-Objekt“ verbunden werden.

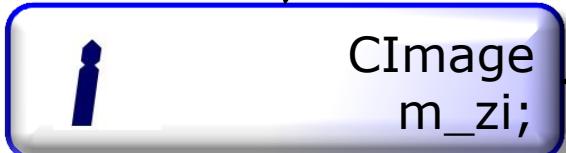
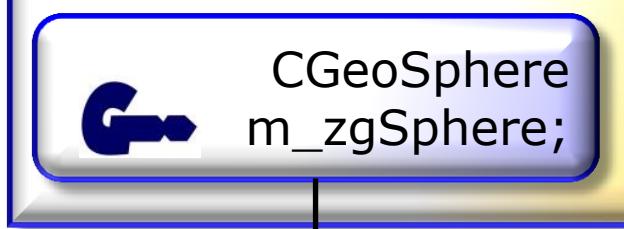
Wir nehmen dazu unsere „Hallo Welt!“-Textur.



„Hallo texturierte Kugel!“-Anwendung

Objekthier. der texturierten Kugel!

Schon bekannte Objekthierarchie



Diffuser
Texturshader

**Hallo
Welt!**

HalоВelt.png





„Hallo texturierte Kugel!“-Anwendung

CMaterial::MakeTextureDiffuse

1 // / / / Anders als in den meisten anderen Engines bzw. Szenegrafen, braucht man sich in Vektoria erst einmal nicht zu kümmern um:

- Shaderprogrammierung
- Materialbeschreibungsskripte
- Verlinkung von Materialien, Texturen und Bildern
- UV-Mapping

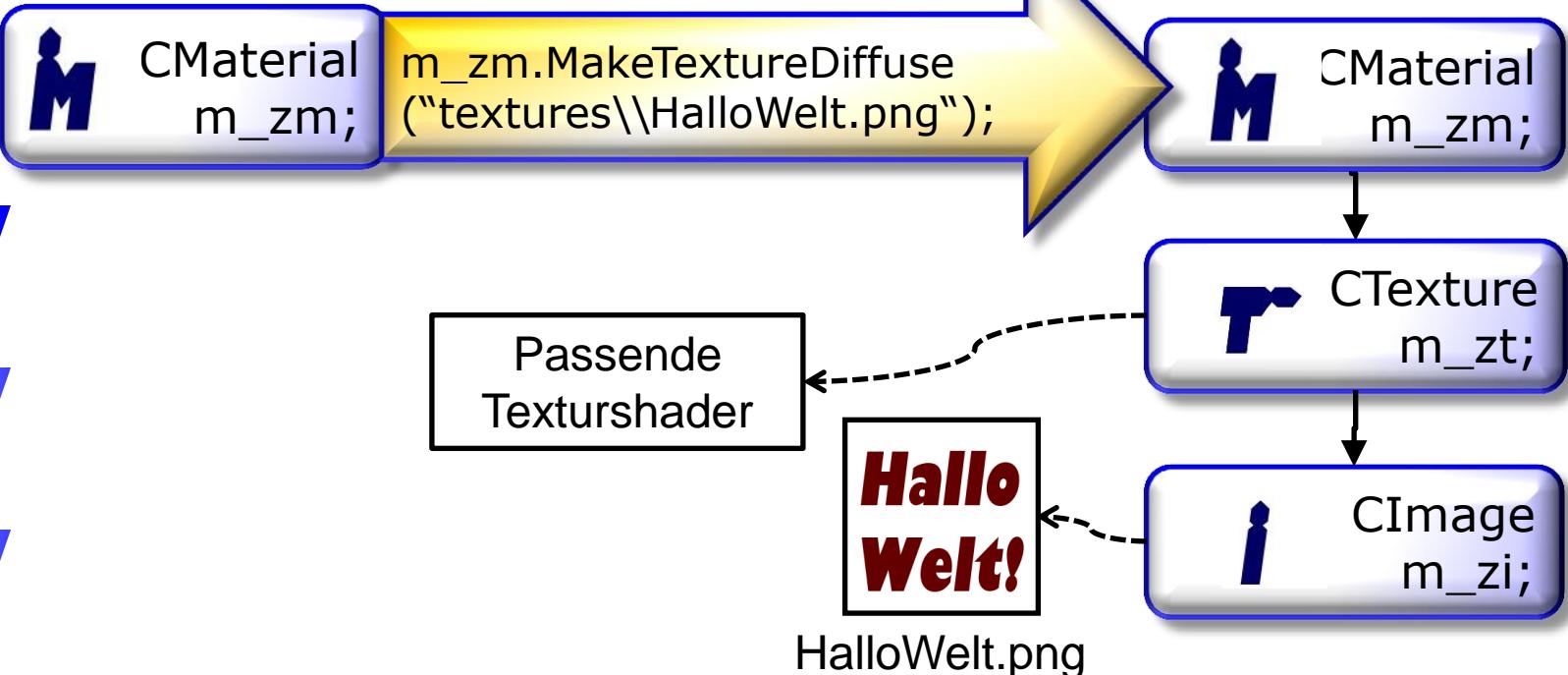
2 // / / / Dies macht die Texturierung extrem einfach und ist eine der Stärken von Vektoria. Man kann die obigen Parameter natürlich trotzdem verändern, wenn man will.

3 // / / / 4 // / / / 5 // / / / Aber fürs Erste reicht die Methode **MakeTextureDiffuse** des Knotenobjektes **CMaterial**.



CMaterial: Die Materialklasse in Vektoria

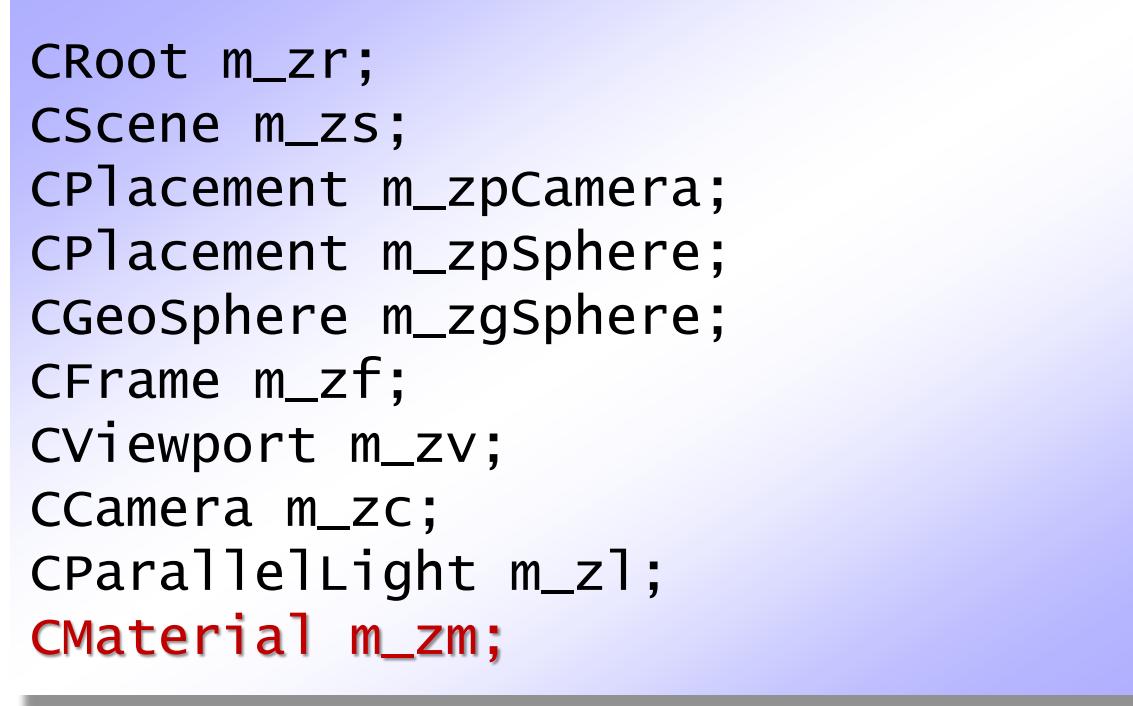
Die Schnelltexturierungsmethoden der Klasse CMaterial erzeugen mit einem einzigen Befehl einen ganzen Hierarchiezweig im Hintergrund:





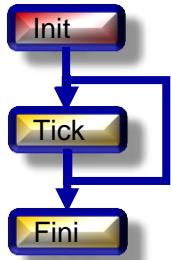
„Hallo texturierte Kugel!“-Anwendung

Vektoria-Objekte in CGame.h (1/3)



```
CRoot m_zr;
CScene m_zs;
CPlacement m_zpCamera;
CPlacement m_zpSphere;
CGeoSphere m_zgSphere;
CFrame m_zf;
CViewport m_zv;
CCamera m_zc;
CParallelLight m_zl;
CMaterial m_zm;
```



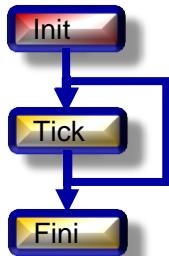


„Hallo texturierte Kugel!“-Anwendung

Init-Methode in CGame.cpp (2/3)

```
void CGame::Init(HWND hwnd, CSplash * psplash)
{
    m_zr.Init(psplash);
    m_zc.Init(QUARTERPI);
    m_zf.Init(hwnd);
    m_zv.InitFull(&m_zc);
    m_zl.Init(CHVector(1.0f, 1.0f, 1.0f),
               CColor(1.0f, 1.0f, 1.0f));
    m_zgSphere.Init(1.5F, &m_zm, 50, 50);
```





„Hallo texturierte Kugel!“-Anwendung

Init-Methode in CGame.cpp (3/3)

```
m_zr.AddFrameHere(&m_zf);  
m_zf.Addviewport(&m_zv);  
m_zr.AddScene(&m_zs);  
m_zs.AddPlacement(&m_zpSphere);  
m_zs.AddPlacement(&m_zpCamera);  
m_zs.AddParallelLight(&m_zl);  
m_zpCamera.AddCamera(&m_zc);  
m_zpSphere.AddGeo(&m_zgSphere);  
m_zm.MakeTextureDiffuse  
    (“textures\\Hallowelt.png”);  
m_zpCamera.TranslateZ(8.0f);  
}
```

1 // / / /

2 // / / /

3 // / / /

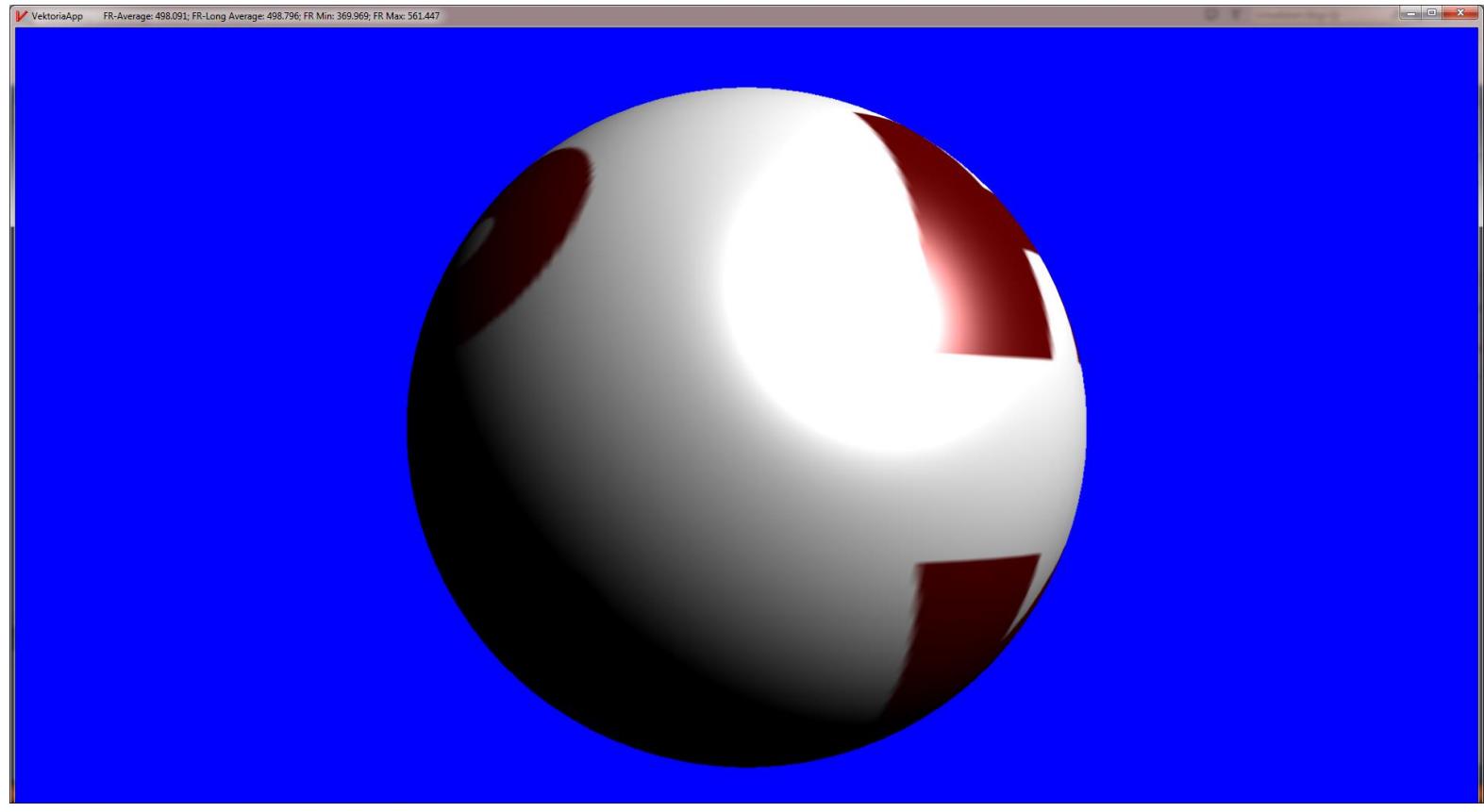
4 // / / /

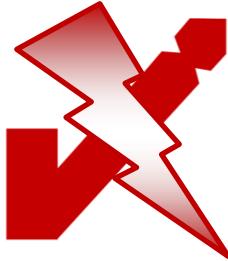
5 // / / /



„Hallo texturierte Kugel!“-Anwendung

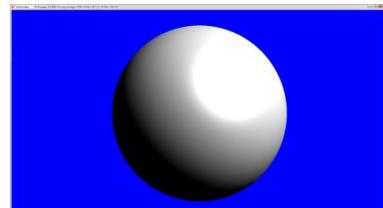
Ausgabe



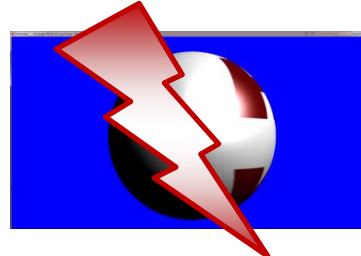


„Hallo texturierte Kugel!“-Anwendung

Weiße Kugel?



statt



?



2 Falls keine Textur auf der Kugel zu sehen ist und die Kugel immer noch weiß aussieht, dann liegt es daran, dass die Textur im angegebenen Ordner nicht gefunden wird.



3 Lösung:

- Schaue im entsprechenden Ordner nach, ob die Textur überhaupt vorhanden ist!
- Überprüfe jeden einzelnen Buchstaben des Texturpfades!
- Checke die Dateiendung! Ist vielleicht **.jpg** statt **.png** angegeben?
- Achte darauf, dass Du im Texturpfad auch **** statt **** angegeben hast!





Hello Kugel-Anwendung

Die Methode WindowReSize

1 // / / / Die Methode **WindowReSize** der Game-Klasse wird immer dann aufgerufen, wenn der Benutzer das Ausgabefenster neu skaliert.

2 // / / / Dort können Sie später die Auflösung der veränderten Fenstergröße anpassen.

3 // / / / Die wichtigste Funktion zum Verändern der Auflösung heißt

4 // / / / **void ReSize(int iHorizontal, int iVertical),**
gehört der Klasse CFrame an.

5 // / / /





Hello Kugel-Anwendung

ReSize



1 // / / / Ziehen Sie das Fenster breiter oder höher! Sie werden bemerken, dass die Kugel nun etwas oval aussieht, dies liegt daran, dass die neuen Fensterausmaße nicht an m_zf, die Objektinstanz von CFrame, heruntergereicht werden. Somit arbeitet m_zf immer noch mit den alten Ausmaßen.

2 // / / / Sie müssen daher in der Methode WindowsReSize der Klasse CGame die neue Breite und Höhe an das Frame herunterreichen:

3 // / / /

```
void CGame::windowResize  
    (int iNewwidth, int iNewheight)  
{  
    m_zf.ReSize(iNewwidth,iNewheight);  
}
```

4 // / / /

5 // / / /





Übung „Hallo Erde!“



Mappen Sie eine Erdtextur auf die Kugel aus dem Hallo Kugel-Programm!

Freiwillige Übung für die schnellen Nerds: Erzeugen Sie mit Photoshop eine eigene sphärische Textur!

Hinweis:
Speichern Sie Ihr Hallo-Kugel-Programm mit der Erdtextur ab. Sie werden es in einer späteren Übung wieder brauchen.





Lösung zur Hallo Erde-Übung

Init-Methode in CGame.cpp



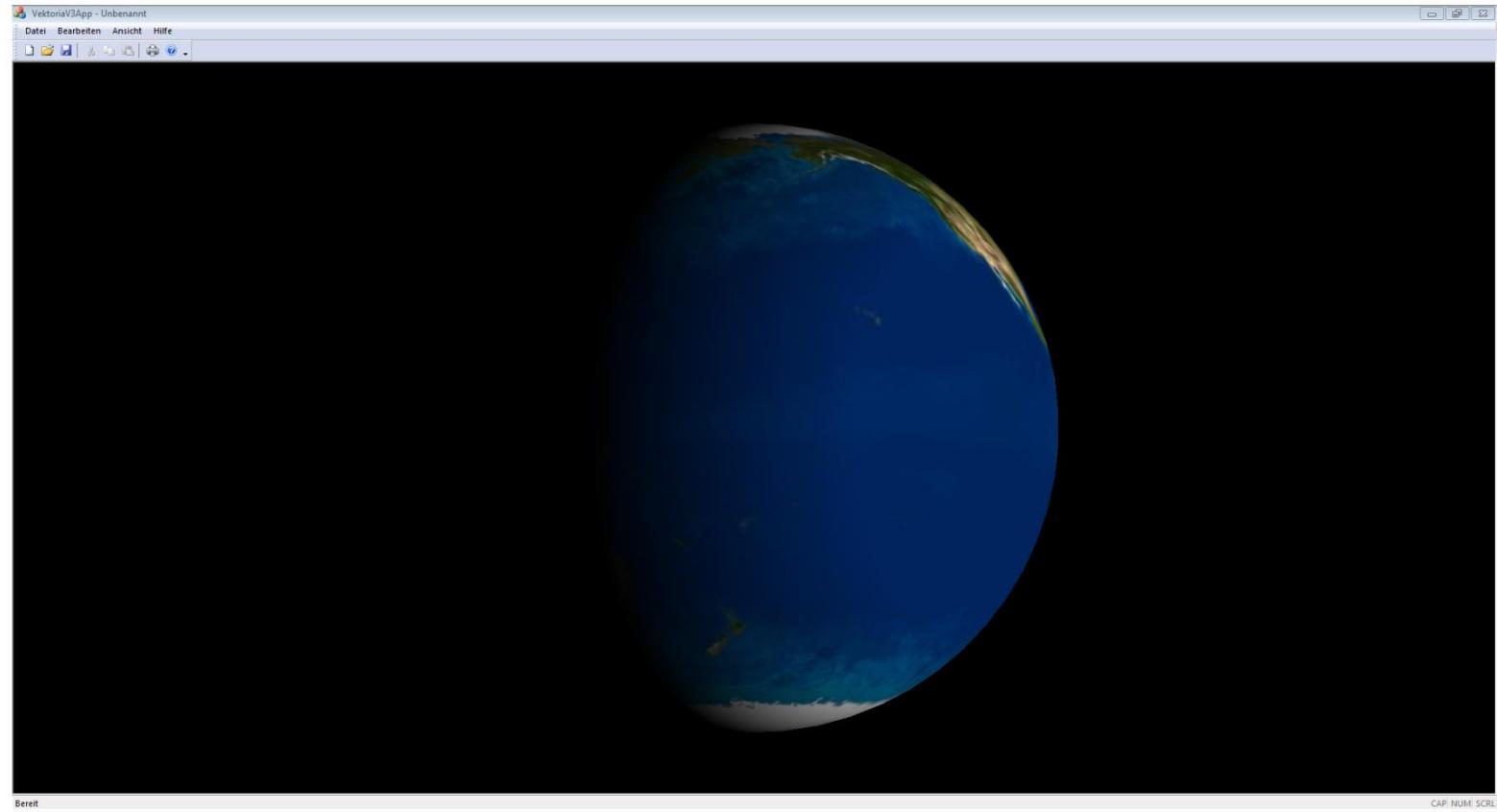
```
void CGame::Init(HWND hwnd, Csplash * psplash)
{
    m_zr.Init(psplash);
    m_zf.Init(hwnd);
    m_zc.Init();
    m_zv.InitFull(&m_zc);
    m_zl.Init(CHVector(1,1,1), ccolor(1,1,1));
    m_zm.MakeTextureDiffuse
        ("textures\\earth_image.jpg");
    m_zr.AddFrameHere(&m_zf);
    m_zf.Addviewport(&m_zv);
    ...
}
```

- 1 // / / /
- 2 // / / /
- 3 // / / /
- 4 // / / /
- 5 // / / /



Lösung zur „Hallo Erde!“-Übung

Ausgabe



1 // / / /

2 // / / /

3 // / / /

4 // / / /

5 // / / /





Übung zu Cameras



Öffnen Sie den Kamerawinkel auf 120°!



Freiwillige Zusatzaufgabe für die schnellen Nerds:



Lassen Sie die (Erd-)Kugel durch die Änderung der NearClippingPlane verschwinden!



(Vorab: Es wird nicht funktionieren!)





Lösung zu Cameras



Öffnen Sie den Kamerawinkel auf 120°!

```
1 /////
2 /////
3 /////
4 /////
5 /////
void CGame::Init(HWND hwnd, Csplash * psplash)
{
    m_zr.Init(psplash);
    m_zf.Init(hwnd);
    m_zc.Init(2.094F); // 2.094F entspricht
        // 120 Grad im Bogenmaß.
    m_zv.InitFull(&m_zc);
    m_zl.Init(CHvector(1,0,1),ccolor(1,1,1));
    m_zm.MakeTextureImage
        ("textures\\earth_image.jpg");
    m_zr.AddFrameHere(&m_zf);
    m_zf.Addviewport(&m_zv);
    ...
}
```





Übung zu Cameras



Lassen Sie den Kamerawinkel während der Laufzeit zwischen 20° und 120° hin- und herschwanken!

Freiwillige Zusatzaufgabe für die schnellen Nerds:

Lassen Sie den Kamerawinkel während der Laufzeit zwischen 20° und 120° wellenförmig (sinoid) hin- und herschwanken!





Lösung zur Camera-Übung

Nerd-Lösung



```
void CGame::Tick( float fTime,
                  float fTimeDelta)
{
    m_zc.SetFov(1.221F+0.872F*sin(fTime));
    // 1,221F entspricht ~70 Grad im Bogenmaß.
    // Dies entspricht dem Mittelpunkt,
    // um den der Kamerawinkel pendelt.
    //
    // 0,872F entspricht ~50 Grad im Bogenmaß.
    // Als Sinusfaktor entspricht dies
    // der halben Amplitude.
    m_zf.Tick(fTimeDelta);
}
```

1 // / / /

2 // / / /

3 // / / /

4 // / / /

5 // / / /



Kapitel 4

Kapitel 4

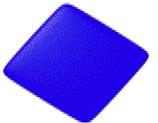


1 // / / /

2 // / / /

3 // / / /

/// VIRTUELL /// VS. /// REAL

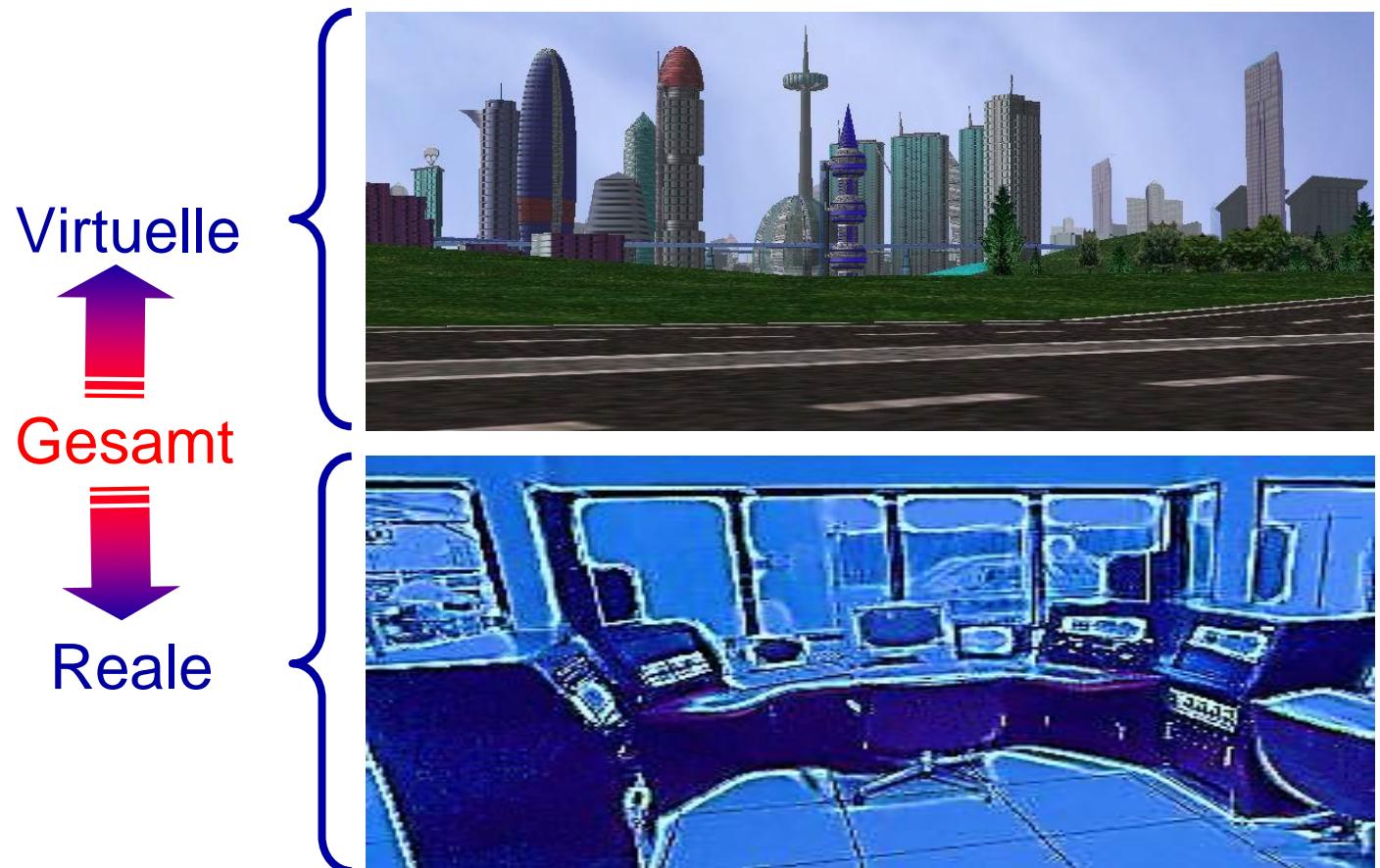


5 // / / /



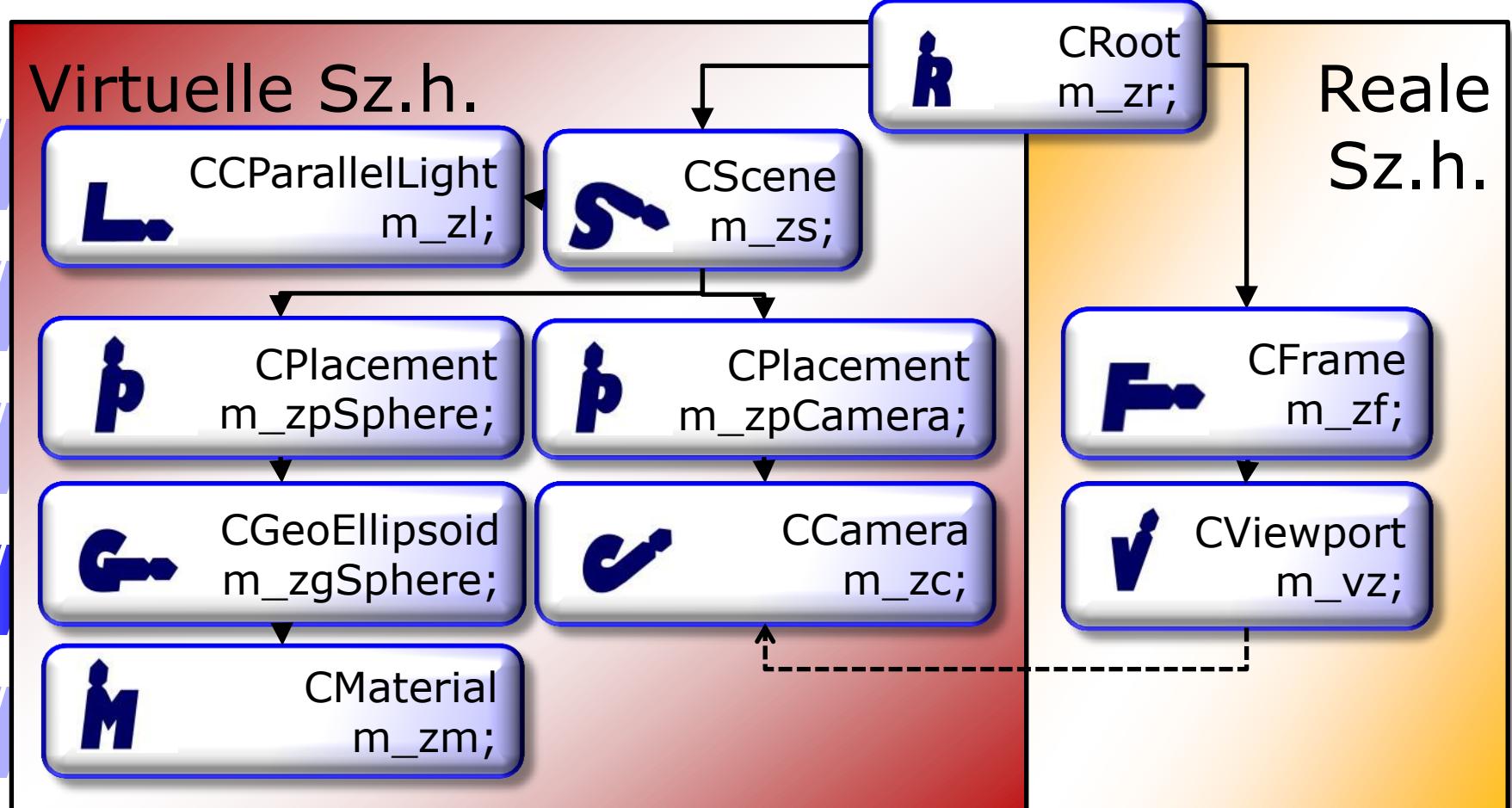
Hallo Kugel

Virtuelle vs. Reale Szenenhierarchie



Hallo Kugel-Anwendung

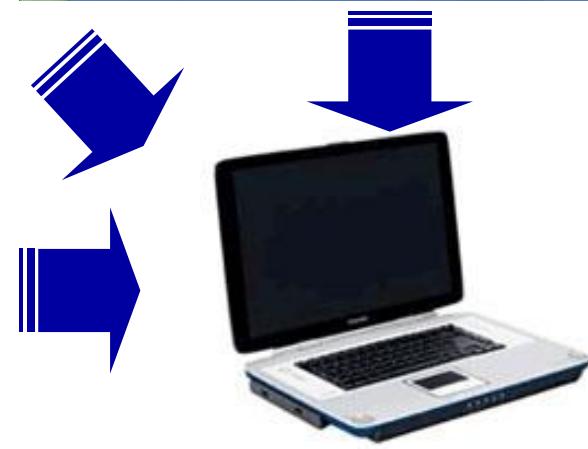
Virtuell vs. Real – Beispiel



Bildung der Objekthierarchie

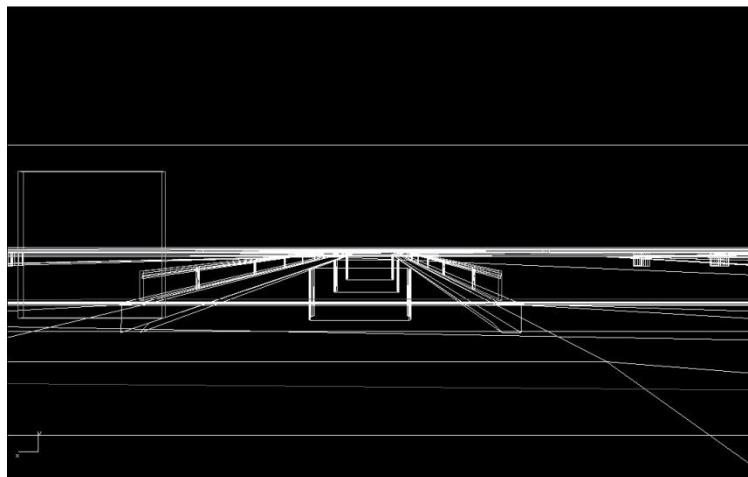
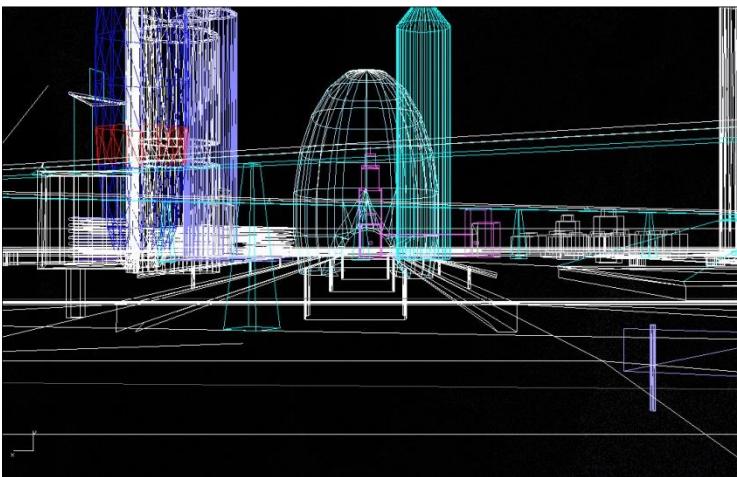
Virtuelle vs. Reale Hierarchie

- 1 // / / /
- 2 // / / /
- 3 // / / /
- 4 // / / /
- 5 // / / /



Optische v.SH. v.SH.

Haptische



1 // / / /

2 // / / /

3 // / / /

4 // / / /

5 // / / /



Kapitel 5

Kapitel 5



1 // / / /

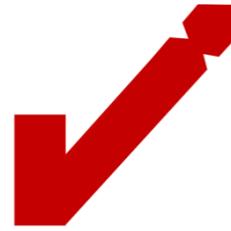
2 // / / /

3 // / / /

4 // / / /

VEKTORIA-NOTATION



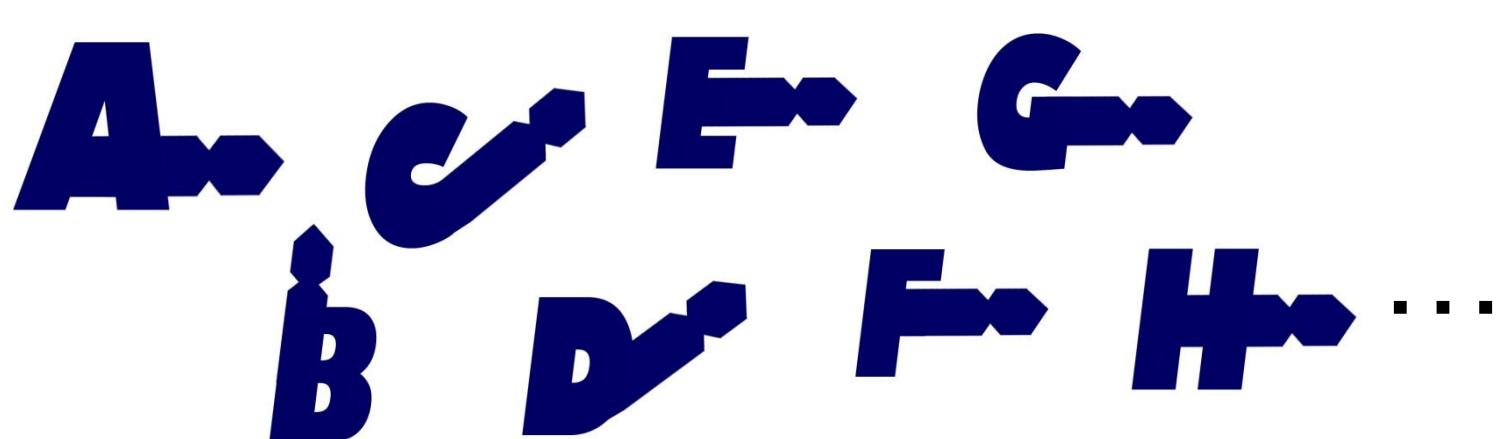


Vektorianische Notation

Vektorianische Notation

Wie ersichtlich, folgen Vektoria-Programme einer speziellen Notation, die am Anfang etwas ungewohnt erscheint.

Diese Notation erspart aber viel Schreib- und Sucharbeit und ist einmal die Grundstruktur von Vektoria verinnerlicht, so fällt es dank der vektorianischen Notation leicht, sich in neue Teile der komplexen Engine einzuarbeiten.





Vektorianische Notation

Anfangsbuchstaben der Knotenobjekte

Die Klassen in Vektoria, die Hauptknotenobjekte repräsentieren, fangen alle mit einem anderen Buchstaben an:



A: CAudio
B: CBackground

C: CCamera
D: CDevice

E: CEmitter

F: CFrame

G: CGeo

H: CHardware

I: CImage

J: (*frei*)

K: CKeyframe (*angedacht*)

L: CLight

M: CMaterial

N: CNode
O: COverlay
P: CPlacement
Q: (*frei*)
R: CRoot
S: CScene
T: (*frei*)
U: CUnion (*angedacht*)
V: CViewport

W: CWriting
X: (*frei*)
Y: (*frei*)
Z: (*frei*)



Vektorianische Variablennotation

Die Buchstabenunterschiedlichkeit in Vektoria ist kein Zufall!
Damit lässt sich gezielt Schreibarbeit durch leicht zu merkende
mnemotechnische Abkürzungen vermindern.

Zum Beispiel können die Präfixe für instanzierte Variablen in
Vektoria immer mit einem „z“ und dem nachfolgenden
Anfangsbuchstaben des Knotenobjektes abgekürzt werden:

CAudio za;
CBackground zb;
CCamera zc;
CDevice zd;
CEmitter ze;
CFrame zf;
etc.





Pluralklassen

Für jede Singular-Klasse gibt es in Vektoria eine Plural-Klasse, die als Container dient. Die Pluralklassen sind hinsichtlich Speicherbedarf und Geschwindigkeit besonders effizient, denn sie haben Smart Allocation implementiert, eine einzigartige Technologie, von Vektoria UG (haftungsbeschränkt). Pluralklassen enden mit einem „s“:

CAudios za

CBackgrounds zbs;

CCameras zcs:

CDevices zds;

CEmitters zde:

CFrames zfs:

etc

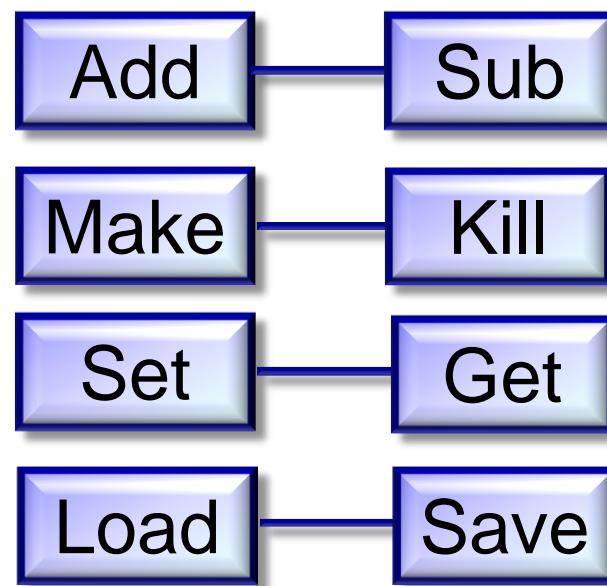
etc.

**PROF. DR. TOBIAS BREINER
VEKTORIA MANUAL**

Vektorianische Methoden-Symmetrie

Wie Sie wahrscheinlich schon erkannt haben, besitzen viele Methoden in Vektoria eine „Symmetrie“.

Zu bestimmten Methoden existiert eine inverse Gegenmethode. Die symmetrischen Methoden sind:





Methoden-Symmetrie Add-Sub-Symmetrie



1 Hängt vorhandenes Objekt an Elternobjekt an.

Hängt vorhandenes Objekt an Elternobjekt ab.



Sub hängt ein Objekt nur ab, es existiert aber danach immer noch im Speicher!





Methoden-Symmetrie

Make-Kill-Symmetrie



1 // / / /
Erzeugt Objekt
und hängt es an
Elternobjekt an.

2 // / / /
Hängt Objekt von
Elternobjekt ab und
zerstört es danach.



3 // / / /
4 // / / /
5 // / / /
Kill kann verständlicherweise nur bei mit
Make oder new erzeugten Objekten
verwendet werden! Nicht auf
Membervariablen anwenden => Crash!





Methoden-Symmetrie

Set-Get-Symmetrie



Set wird auch bei Kind-Objekten angewandt, bei denen ein Vater immer nur genau ein Kind haben kann!

Beispiel: Materialien, die an Geometrien angehangen werden (Ein Geometrie hat immer genau ein Material und niemals mehr).





Methoden-Symmetrie

Load-Save-Symmetrie



1 // / / /
Lädt Parameter für
die spezifische
Klasse aus einer
Datei.

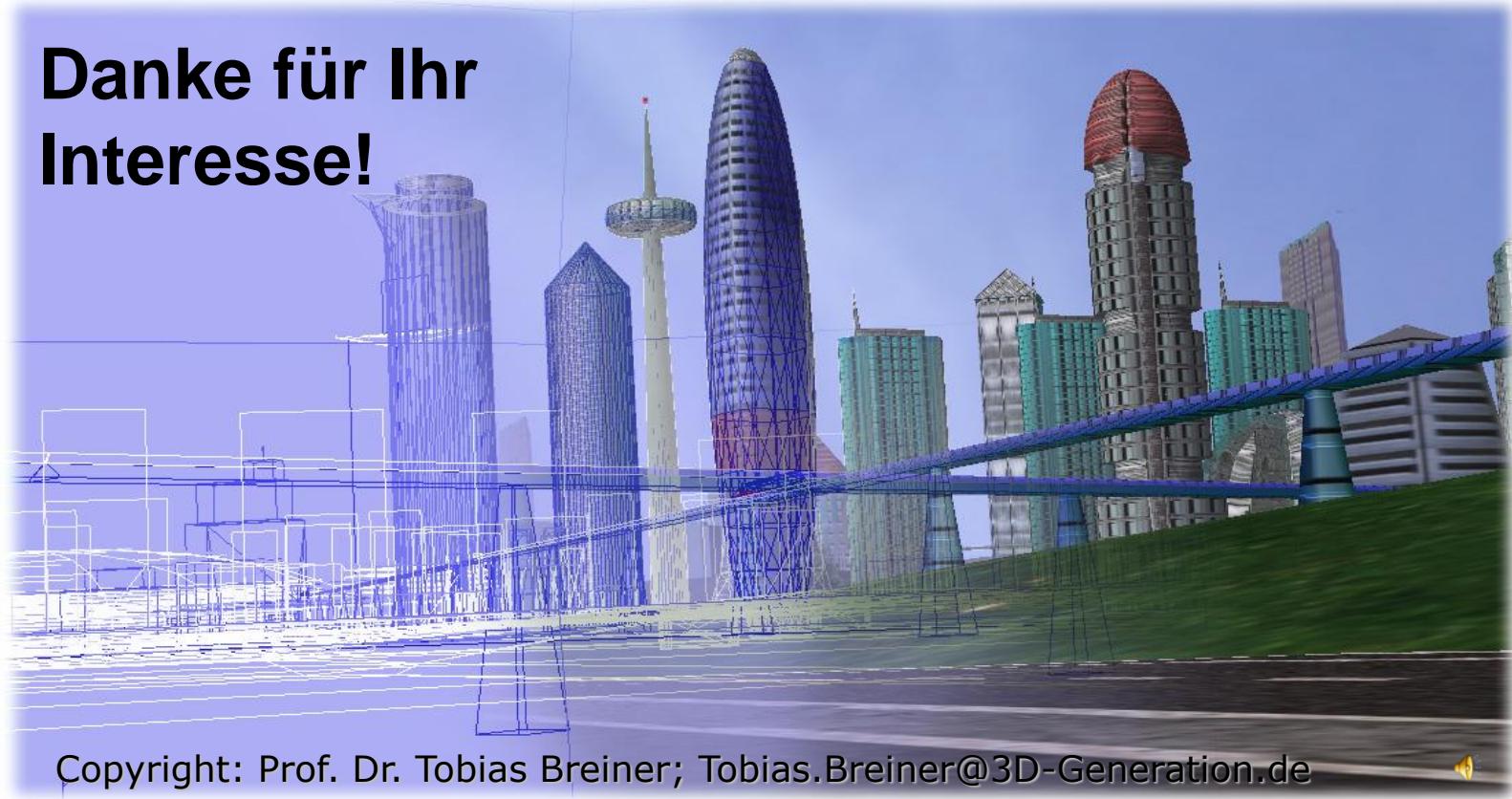
2 // / / /
Speichert Parameter
für die spezifische
Klasse aus einer
Datei.

3 // / / /
4 // / / /
5 // / / /



|||||GAME ||||OVER

Danke für Ihr
Interesse!



Copyright: Prof. Dr. Tobias Breiner; Tobias.Breiner@3D-Generation.de

