Dynamische Texturen mit P-Buffers

Kein Licht ohne Schatten

Viele **Techniken der Computergrafik** wie das Rendering von Schatten oder prozedurale Texturen benötigen dynamisch erzeugte Texturen. Wir zeigen, wie Sie dynamische Texturen in OpenGL effizient rendern.

CARSTEN DACHSBACHER

3D-Engines und Computerspiele enthalten oft dynamisch erzeugte (gerenderte) Texturen. Mit dieser Technik können Sie Shadow Maps, Feedback-Effekte, Impostors, Dynamische Cube/Environment Maps und viele andere Dinge darstellen.

In dieser Ausgabe lernen Sie P-Buffers kennen und erfahren, wie Sie diese effizient einsetzen. Ein P-Buffer ist ein *Off Screen* (nicht sichtbarer) Pixel Buffer, der einen eigenen OpenGL-Kontext besitzt. Zu einem Kontext gehören alle Einstellungen der OpenGL States wie Matrizen, Materialparameter, Lichtquellen und Texturen.

Der Vorteil eines P-Buffers ist, dass seine Auflösung und sein Pixel-Format unabhängig vom aktuellen Darstellungsmodus sind. Hingegen ist der On-Screen-Buffer, also der normale Rendering Buffer, an die Bildschirm-Auflösung und -Farbtiefe gebunden.

In einen P-Buffer können Sie genauso rendern wie in einen On-Screen-Buffer und dazugehörigen OpenGL-Kontext. Um P-Buffers zu verwenden, muss Ihre OpenGL-Implementation die Erweiterung *WGL_ARB_pixel_format* und *WGL_ARB_pbuffer* unterstützen.

Einen P-Buffer anlegen

Zunächst benötigen Sie die Zeiger auf die Funktionen, welche die erwähnten OpenGL Extensions definieren. Diese Zeiger holen Sie sich mit dem Befehl *wglGetProcAdress* während der Initialisierung. Die benötigten Funktionen finden Sie in der Tabelle auf Seite 170.

Nun legen Sie den P-Buffer an. Weil ein Programm mehrere P-Buffers einsetzen kann, definieren Sie eine Klasse, die Ihnen Arbeit abnimmt und alle benötigten Informationen speichert:

```
class CPBuffer
{ private:
    HPBUFFERARB hPBuffer;
    HDC         hDC;
    HGLRC       hRC;
    // Größe
    int         sizeX, sizeY;
    // Texture
    GLuint      textureID;
    // Status
    int         exists;
  public:
    CPBuffer( int _x, int _y,
              HDC hDC );
    ~CPBuffer();
    int     bind();
    int     release();
    int     makeCurrent();
    GLuint  getTexID()
      { return textureID; };
};
```

Den schwierigsten Teil stellt der Konstruktor dar, der den P-Buffer erzeugt. Zunächst müssen Sie festlegen, welches Pixel-Format der P-Buffer hat. Bei dieser Klasse legen Sie immer einen Buffer mit folgenden Parametern (mit Null terminiert) an, die Sie beliebig für jeden einzelnen P-Buffer modifizieren können:

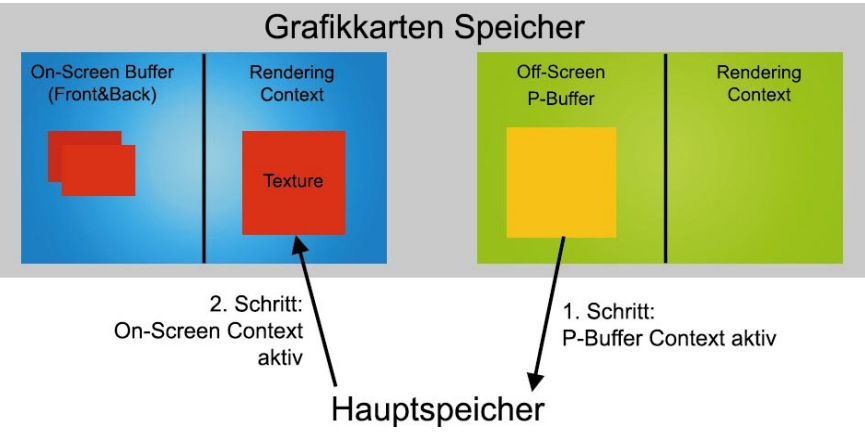
```
hPBuffer = NULL;
sizeX    = _x;
sizeY    = _y;
int pfAttribute[] =
{
  // Verwendung von OpenGL
  WGL_SUPPORT_OPENGL_ARB, TRUE,
  WGL_DRAW_TO_PBUFFER_ARB, TRUE,
// P-Buffer als Texture
WGL_BIND_TO_TEXTURE_RGBA_ARB,
                          TRUE,
  // RGBA 8888 Format
  WGL_RED_BITS_ARB, 8,
  WGL_GREEN_BITS_ARB, 8,
  WGL_BLUE_BITS_ARB, 8,
  WGL_ALPHA_BITS_ARB, 8,
  // >16 Bit Z-Buffer
  WGL_DEPTH_BITS_ARB, 16,
  // kein Double Buffer
  WGL_DOUBLE_BUFFER_ARB, FALSE, 0
};
```

Dann überprüfen Sie, ob ein solches Format unterstützt wird. Mit *wglChoosePixelFormatARB* können Sie eine Liste von Pixel-Formaten anfordern, mit der sie die notwendigen Parameter am besten an Ihren Programmlauf anpassen. Wollen Sie nur ein Format erhalten, verwenden Sie den Aufruf

```
int pixelFormat, nFormat = 0;

wglChoosePixelFormatARB(
  _hDC, pfAttribute, NULL, 1,
  &pixelFormat, &nFormat );
if ( nFormat == 0 )
  kein passendes Format !
```

Ist das Pixel-Format bestimmt, können Sie die Verwendung des P-Buffers festlegen. Um ihn als dynamische 2D-Tex-



KOPIEREN SIE DEN P-BUFFER-INHALT über den Systemspeicher in eine Textur.

AUF CD

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Heft Add-ons/Programmierung/PC Underground.*

tur zu verwenden, definieren Sie folgendes Parameter-Array:

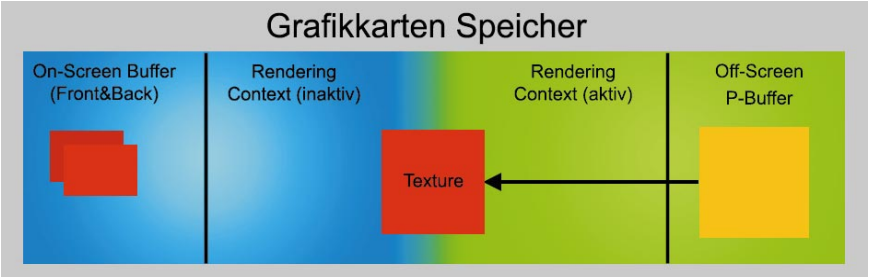
```
int pbAttribute[] =
{  // Texture Format RGBA 8888
  WGL_TEXTURE_FORMAT_ARB,
  WGL_TEXTURE_RGBA_ARB,
  // 2D Texture
  WGL_TEXTURE_TARGET_ARB,
  WGL_TEXTURE_2D_ARB, 0 };
```

und erzeugen den P-Buffer:

```
hPBuffer = wglCreatePbufferARB(
  _hDC, pixelFormat,
  sizeX, sizeY, pbAttribute );
hDC = wglGetPbufferDCARB
                  ( hPBuffer );
hRC = wglCreateContext( hDC );
if( !hPBuffer )
  P-Buffer nicht ok !
```

Zuletzt überprüfen Sie, ob der P-Buffer korrekt angelegt wurde, indem Sie seine Größe mit der gewünschten – gegeben durch die Konstruktorparameter *_x* und *_y* – vergleichen:

```
int __x, __y;
wglQueryPbufferARB( hPBuffer,
 WGL_PBUFFER_WIDTH_ARB,  &__x );
wglQueryPbufferARB( hPBuffer,
 WGL_PBUFFER_HEIGHT_ARB, &__y );
if( !(__x==sizeX && __y==sizeY))
  Größe nicht ok !
```



KOPIEREN SIE DEN P-BUFFER-INHALT mit Shared Textures.

Rendering mit P-Buffers

Um in den P-Buffer rendern zu können, müssen Sie dessen Kontext als den aktuellen *OpenGL Rendering Context* wählen. Zu einem OpenGL-Kontext gehören alle Einstellungen zu Matrizen, Kamera, Texturen, Lichtquellen etc.

Die Einstellungen für den P-Buffer können und müssen unabhängig vom normalen (sichtbaren) Rendering-Kontext gesetzt werden. Befehle, die solche OpenGL States modifizieren, werden immer auf den aktuell gewählten Kontext angewendet. Den Kontext eines P-Buffers wählen Sie mit der Methode *makeCurrent()* der obigen Klasse aus. Diese sieht wie folgt aus:

```
int CPBuffer::makeCurrent()
{ // P-Buffer angelegt ?
  if ( !exists ) return 0;
  // Auswählen
if( !wglMakeCurrent( hDC, hRC ))
    return 0;
  return 1; }
```

Wenn Sie diese Methode aufrufen, beziehen sich alle darauf folgenden OpenGL-Aufrufe nur noch auf den P-Buffer. Diesen können Sie mit *glClear(...)* löschen und rendern. Um auf den normalen On-Screen-Kontext

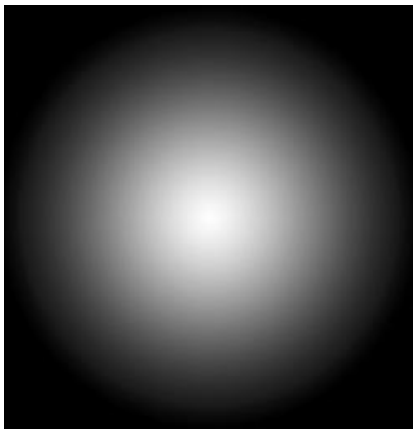
FUNKTIONEN VON WGL_ARB_PIXEL_FORMAT UND WGL_ARB_PBUFFER

Funktion	Erläuterung
wglChoosePixelFormatARB	Abfrage/Auswahl eines Pixel-Formats
wglCreatePbufferARB	Erzeugen eines Pixel-Buffers, liefert Handle zurück
wglGetPbufferDCARB	Erzeugt ein Device Context für einen P-Buffer
wglReleasePbufferDCARB	Gibt obigen Device Context wieder frei
wglDestroyPbufferARB	Zerstört P-Buffer
wglQueryPbufferARB	Abfrage von Breite/Höhe des P-Buffers und ob der P-Buffer nach einer Änderung der Bildschirmauflösung noch existiert

zurückzukommen, rufen Sie die *wglMakeCurrent*-Funktion mit dem Device Context und dem OpenGL Rendering Context auf, den Sie für das normale OpenGL-Fenster erzeugt haben. Diese Initialisierung übernimmt der OpenGL Framework Code für Sie.

Dynamische Texturen

Die bisher vorhandene Funktionalität erlaubt es Ihnen, einen P-Buffer mit beliebigem Format anzulegen und etwas



DER HELLIGKEITSVERLAUF einer Punktlichtquelle

darauf zu rendern. Jetzt können Sie dessen Inhalt als Textur für den On-Screen-Context verwenden. Allerdings müssen Sie den Inhalt des P-Buffers in den Systemspeicher kopieren und als Textur wieder dem anderen Kontext übergeben. Der Ablauf laut der Darstellung des vorigen Bilds entspricht dann dem Code:

```
// Initialisierung
CPBuffer *pBuffer =
  new pBuffer( 512, 512, hDC );
pBuffer->makeCurrent();
... // Rendern auf den P-Buffer
// kopieren des P-Buffer Inhalts
GLubyte data[ 512*512*4 ];
glReadPixels( 0, 0, 512, 512,
   GL_RGBA, GL_UNSIGNED_BYTE,
                        data );
// On-Screen Context
wglMakeCurrent( os_hDC, os_hRC);
// Texture generieren & upload
glGenTextures( 1, &tID );
glBindTexture
          ( GL_TEXTURE_2D, tID );
glTexImage2D(  GL_TEXTURE_2D, 0,
  GL_RGBA, 512, 512, 0, GL_RGBA,
       GL_UNSIGNED_BYTE, data );
```

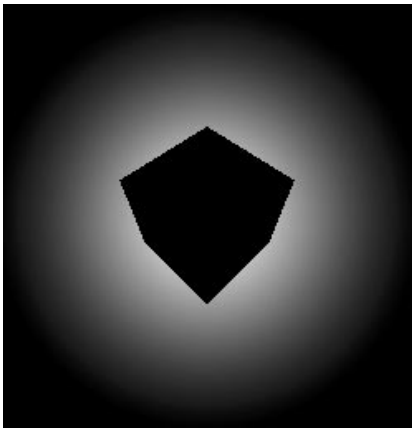
Diese Vorgehensweise ist nicht sehr schnell. Mit dem *glReadPixels*- und dem *glTexImage2D*-Befehl kopieren Sie Daten aus dem Speicher der Grafikkarte und anschließend wieder hinein. Diesen Aufwand können Sie vermeiden.

Mit OpenGL können Sie die Display Lists und Texturen für verschiedene Rendering Contexts gemeinsam nutzen. Diese Variante gestattet es Ihnen, eine Textur anzulegen und in diese direkt den Inhalt des P-Buffers mit *glCopyTexSubImage2D* zu kopieren.

Die Contexts müssen die Texturen gemeinsam nutzen. So aktivieren Sie diese:

```
if ( wglShareLists
      ( os_hDC, pBuffer->hDC ) )
//ok-> gemeinsame Nutzung
```

Das Äquivalent zum obigen Code-Ausschnitt für dynamische Texturen ist dadurch vereinfacht und schneller:



EINE SHADOW MAP für einen Würfel

```
// Initialisierung
CPBuffer *pBuffer = ...;
glGenTextures( 1, &tID );
glBindTexture
          ( GL_TEXTURE_2D, tID );
glTexImage2D(  GL_TEXTURE_2D, 0,
  GL_RGBA, 512, 512, 0, GL_RGBA,
       GL_UNSIGNED_BYTE, data );
// P-Buffer füllen
pBuffer->makeCurrent();
// Texture kopieren
glBindTexture
          ( GL_TEXTURE_2D, tID );
glCopyTexSubImage2D
           ( GL_TEXTURE_2D,
      0, 0, 0, 0, 512, 512 );
// On-Screen Context
wglMakeCurrent( os_hDC, os_hRC);
```

In der obigen Variante ist immer noch der *glCopyTexSubImage2D*-Funktionsaufruf enthalten, der den P-Buffer-Inhalt in eine Textur kopiert.

Doch warum muss er kopiert werden, schließlich befindet er sich schon im Speicher der Grafikkarte und könnte gleich als Textur verwendet werden? Die Antwort: Er muss nichts kopiert werden, sofern Ihre Grafikkarte die Erweiterung *WGL_ARB_render_texture* unterstützt.

Die P-Buffer-Klasse enthält noch zwei nicht spezifizierte Methoden, die genau für diesen Zweck gedacht sind. Es ist nämlich möglich, den P-Buffer an eine Textur eines anderen Kontexts zu binden. Das bedeutet, der Inhalt der Textur muss nicht vom Systemspeicher oder einem OpenGL-Kontext kopiert werden, sondern ist automatisch der Inhalt des P-Buffers.

Das Anbinden erfolgt mit der *bind()*-Methode:

```
int CPBuffer::bind()
{ if ( !exists ) return 0;
  if( !wglBindTexImageARB(
         hPBuffer,
         WGL_FRONT_LEFT_ARB ) )
     return 0;
  return 1;   }
```

Genauso müssen Sie die Verbindung einer Textur wieder aufheben können, weil Sie sonst den Inhalt des P-Buffers nicht weiter modifizieren können:

```
int CPBuffer::release()
{ if ( !exists ) return 0;
  if( !wglReleaseTexImageARB(
         hPBuffer,
         WGL_FRONT_LEFT_ARB ) )
    return 0;
  return 1,
}
```

Mit diesen Methoden sieht die Rendering-Schleife nun folgendermaßen aus:

```
// Initialisierung
CPBuffer *pBuffer = ...;
glGenTextures( 1, &tID );
glBindTexture
          ( GL_TEXTURE_2D, tID );
glTexImage2D(  GL_TEXTURE_2D, 0,
  GL_RGBA, 512, 512, 0, GL_RGBA,
       GL_UNSIGNED_BYTE, data );
// P-Buffer füllen
pBuffer->makeCurrent();
// Texture kopieren
glBindTexture
          ( GL_TEXTURE_2D, tID );
pBuffer->bind();
// On-Screen Context
wglMakeCurrent( os_hDC, os_hRC);
//Zeichen: P-Buffer als Textur
pBuffer->release();
```

Mipmapping

Um Aliasing-Effekte für klein dargestellte Texturen zu vermeiden, verwenden Sie Mipmaps, also verkleinerte Abbilder von Texturen. Für Ihre statischen Texturen können Sie diese im Normalfall selbst erzeugen und an OpenGL übergeben, oder Sie gebrauchen *gluBuild2DMipmaps(...)* anstelle von *glTexImage2D(...)*.

Beide Varianten sind allerdings nicht so effizient wie die der Grafik-Hardware. Wenn diese die Erweiterung *SGIS_generate_mipmap* unterstützt, können Sie Mipmap-Stufen von Texturen dynamisch erzeugen lassen. Eine solche Textur legen Sie folgendermaßen an:

```
glGenTextures( 1, &tID );
glBindTexture
        ( GL_TEXTURE_2D, tID );
// Texture Filter
glTexParameteri( GL_TEXTURE_2D,
        GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_NEAREST );
glTexParameteri( GL_TEXTURE_2D,
        GL_TEXTURE_MAG_FILTER,
                  GL_LINEAR );
// Mipmap Generierung ein !
glTexParameteri( GL_TEXTURE_2D,
       GL_GENERATE_MIPMAP_SGIS,
                    GL_TRUE );
glTexImage2D(  GL_TEXTURE_2D, 0,
  GL_RGBA, 512, 512, 0, GL_RGBA,
       GL_UNSIGNED_BYTE, data );
```

Dieser Weg steht Ihnen sogar bei einem dynamisch gebundenen P-Buffer offen. Nachdem Sie Ihre Textur entsprechend programmiert haben, müssen Sie bei der Anlage des P-Buffers vorsorglich ausreichend Speicher für die Mipmaps reservieren. Dazu erweitern Sie die *pbAttribute*-Parameterliste (vor der Nullterminierung) um das folgenden Attribut:

```
WGL_MIPMAP_TEXTURE_ARB,
                  GL_TRUE
```

P-Buffers müssen sich mit dem Frame Buffer, Texturen und Display Lists den Speicher der Grafikkarte teilen. Deshalb sollten Sie sich bei Texturen mit zu üppig dimensionierten Auflösungen und Farbtiefen zurückhalten. Andernfalls begrenzen Sie Ihre Render-Performance rapide. Oft genügt ein einziger P-Buffer, den Sie mehrfach pro Renderpass verwenden. Unter Umständen benötigen Sie für den P-Buffer gar keinen zugehörigen Z-Buffer und können so weiteren Speicher sparen.

Beispiel für P-Buffers

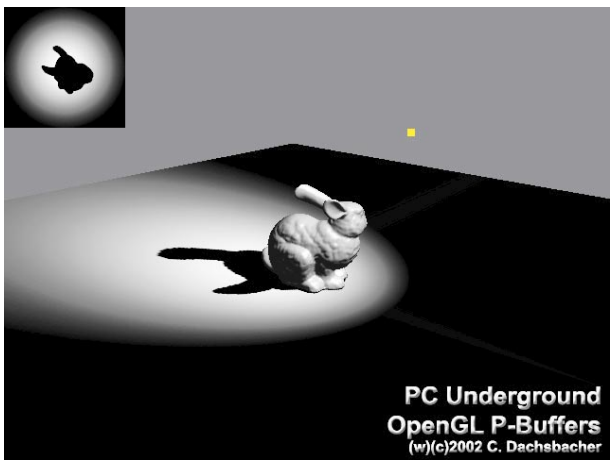
Setzen Sie dynamische Texturen ein, um Schatten darzustellen. Aus der PC-Underground-Serie kennen Sie schon einige Schatten-Rendering-Techniken.

Hier lernen Sie eine einfache Shadow-Map-Technik und deren Umsetzung mit P-Buffers kennen. Einfach bedeutet in diesem Fall, dass Objekte sich nicht selbst beschatten können und dass der Schatten eines Objekts (der auf alle anderen Objekte der Szene geworfen wird) in einer Textur gespeichert ist.

Dazu legen Sie einen P-Buffer mit der oben vorgestellten Klasse an. Wenn Sie die Performance steigern wollen, können Sie sogar den Z-Buffer weglassen.

Die Lichtquelle ist durch ihre Position und die Lichtrichtung gegeben bzw. durch einen Punkt im Raum, in dessen Richtung sie scheint. Um den Schatten eines Objekts bezüglich dieser Punktlichtquelle darzustellen, benötigen Sie diesen in Form einer Textur. Setzen Sie dazu die Kamera des P-Buffer-Kontexts auf die Position der Lichtquelle und der entsprechenden Richtung, und erzeugen Sie eine projektive Abbildung in der Projection Matrix:

```
pBuffer->makeCurrent();
// Transformation Lichtquelle
glMatrixMode( GL_PROJECTION );
glPushMatrix();
glLoadIdentity();
gluPerspective( 80.0f, 1.0f,
                1.0f, 500.0f);
glMatrixMode( GL_MODELVIEW );
glPushMatrix();
glLoadIdentity();
gluLookAt(lightPosition[ 0 ],
lightPosition[ 1 ],
lightPosition[ 2 ], 0, 0, 0,
  0, 1, 0 );
```



UNSER BEISPIELPROGRAMM mit Shadow Maps in P-Buffers

Wenn Ihre Lichtquelle einen Helligkeitsverlauf im Lichtkegel besitzen soll, können Sie diesen in einer Textur wie im Bild oben angeben.

Zeichnen Sie diesen Helligkeitsverlauf zunächst gestreckt über den P-Buffer. Anschließend färben Sie das Schatten werfende Objekt schwarz. Dadurch erhalten Sie die Shadow Map des Objekts, welches Sie im Beispiel des nächsten Bildes betrachten können.

Beim Rendering des eigentlichen Bildes im On-Screen Buffer projizieren Sie die Shadow Map auf alle Schatten empfangenden Objekte der Szene. Zunächst setzen Sie die Kameraparameter:

```
wglMakeCurrent
            ( os_hDC, os_hRC );
// normale Kameratransformation
...
glBindTexture( GL_TEXTURE_2D,
pBuffer->getTexID() );
pBuffer->bind();
```

Anschließend verwenden Sie die OpenGL-Textur-Koordinatengenerierung. Diese berechnet für Sie aus den Vertexkoordinaten der Objekte die Texturkoordinaten bezüglich der Shadow Map. Im ersten Schritt reichen Sie die Weltkoordinaten mit *glTexGen*(...)*-Befehlen direkt als Texturkoordinaten durch. In der Textur-Matrix befinden sich die Transformation und Projektion der Koordinaten, die identisch zu den Lichtquellen-Transformationen sind:

```
float genS[]=
        { 1.0, 0.0, 0.0, 0.0 };
float genT[]=
        { 0.0, 1.0, 0.0, 0.0 };
float genR[]=
        { 0.0, 0.0, 1.0, 0.0 };
float genQ[]=
        { 0.0, 0.0, 0.0, 1.0 };
// analog für T, R, Q (HEFT-CD)
```

Jetzt rendern Sie die Schatten empfangenden Objekte mit den folgenden Textur-Parametern:

```
glTexParameteri(
GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE );

//...
glColor4ub( 255,
255, 255, 255);
renderShadowRecei-
ver();
```

Zum Abschluss zeichnen Sie noch das Schatten werfende Objekt und erhalten ein Resultat, wie Sie es im Bild sehen.

Die P-Buffer-Klasse gestattet also nicht nur einfaches Handling verschiedener RenderTargets, sondern bietet auch einen sehr performanten Zugang, dynamische Texturen zu erzeugen. ET

Weiterführende Websites

www.dachsbacher.de/pcu

www.nvidia.com