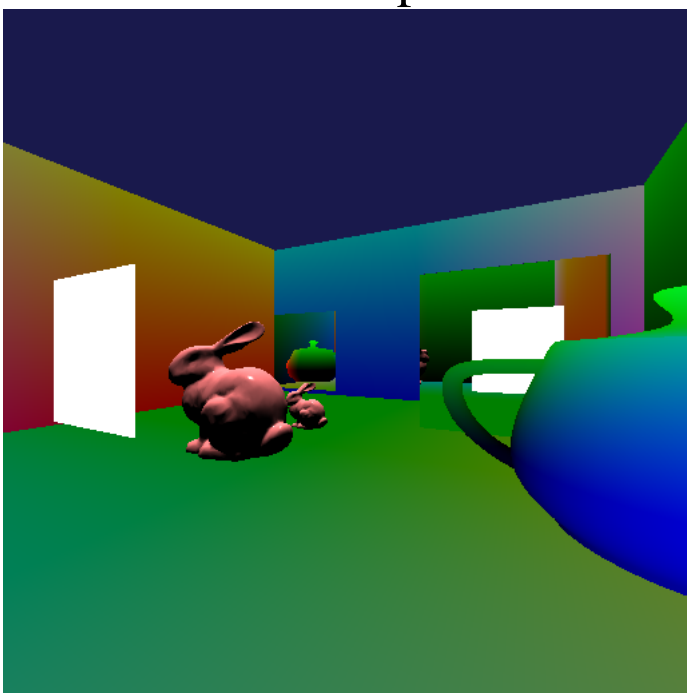
Portaler i OpenGL



Alexander Uggla
aleug359@student.liu.se
TSBK03 : Tekniker för avancerade datorspel

31 januari 2020

Innehåll

Figurer

Kapitel 1

Introduktion

Portaler är ett återkommande fenomen som används inom 3D-datorgrafik. Men hur kan denna portaleffekt skapas? Denna rapport avser visa hur portaler kan implementeras i OpenGL, men teknikerna som presenteras kan även användas i andra sammanhang.

1.1 Definition av portal

En portal definieras som en sömlös koppling mellan två platser. Ett exempel på portaler är portalerna som skapas av *The Portal Gun* i spelet *Portal* skapad av *Valve* [3] som kan ses i figur 1.1. Figuren visar två portalöppningar, gul och blå, som är kopplade till varandra och den vissuella effekten som detta skapar.

Figur 1.1: Portaler från spelet Portal

I denna rapport syftar begreppet portal på två portalöppningar som är kopplade till varandra.

1.2 Krav och mål

Portalerna ska uppfylla följande krav:

- Varje portal ska ha två portalöppningar som går till varandra.
- Det ska vara möjligt att se in genom den ena portalöppningen och ut genom den andra.

- Vyn genom portalen ska ha samma perspektiv som kameran ser på portalöppningen med.
- Det ska inte vara någon skillnad i upplösningen mellan portalen och övriga världen.
- När en kamera går genom den ena portalen ska den sömlöst transporteras till den andra.
- Det ska vara möjligt att se båda portalöppningarna för en portal i samma bild.
- Objekt bakom en portalöppning ska inte synas i den andra portalöppningen.

1.3 Begränsningar

Implementeringen kommer inte behandla rekursiva portaler, det vill säga portalöppningar som syns genom andra portalöppningar. Portalöppningarna kommer dessutom vara statiska i scenen. Objekt som passerar genom portaler kommer inte heller behandlas.

1.4 Plattform och utvecklingsmiljö

Koden kommer skrivas i *C* med *OpenGL* som grafikbibliotek. Utöver detta används *Ingemar Ragnemalms* bibliotek *VectorUtils*[2] för alla linjär algebra-operationer. Utvecklingen kommer ske i operativsystemet *Linux*. Koden kommer skivas i *CLion* och kompileras med *makefile*.

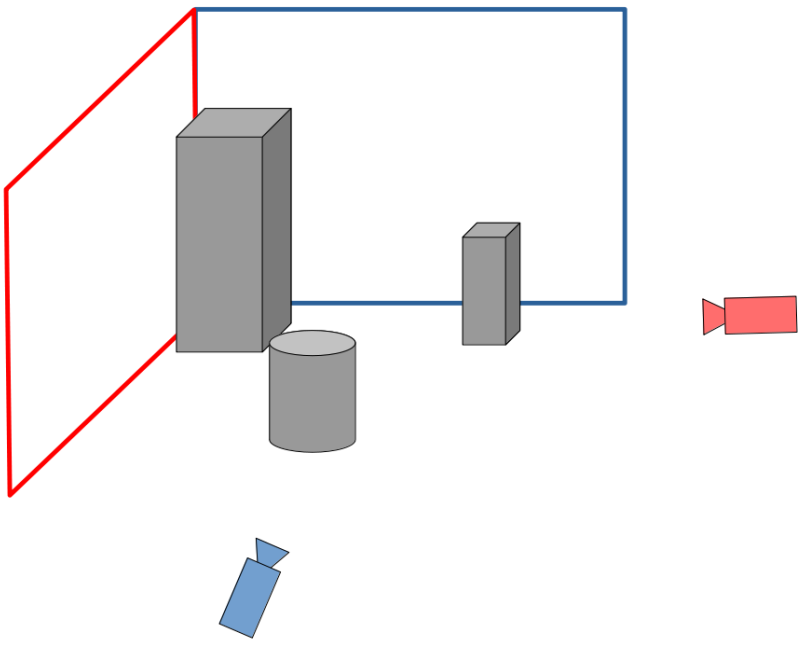
Kapitel 2

Implementation

Portalerna implementeras i OpenGL och utnyttjar ett flertal buffrar som OpenGL tillhandahåller. Specifikt utnyttjas stencilbuffern och djupbuffern för att uppnå portaleffekten.

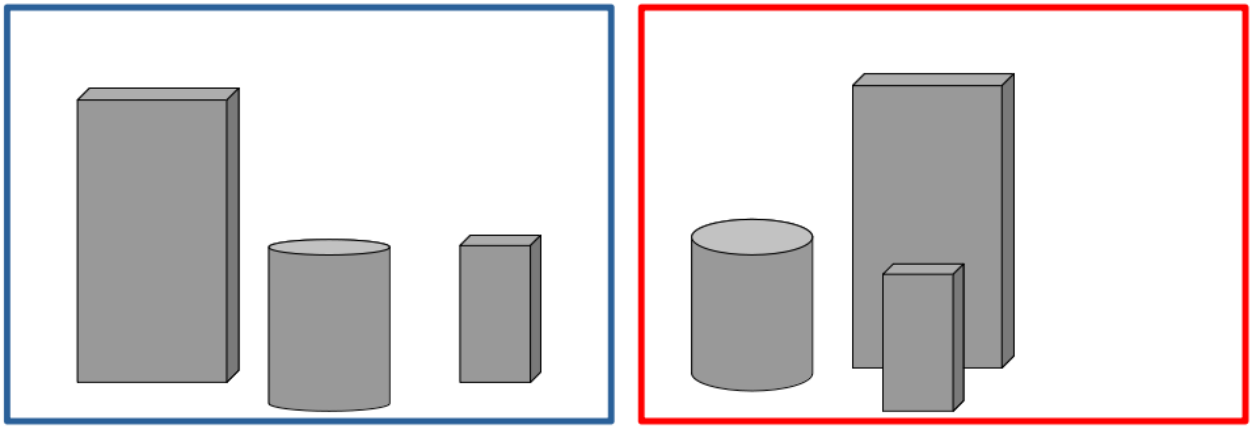
2.1 Rendering till buffrar i OpenGL

I OpenGL renderas inte scener direkt till pixlar, utan har buffrar som mellansteg. När ett objekt ritas ut på skärmen skickas dess vertexdata, vilken shader den använder och övriga parametrar till datorns GPU. Först när alla objekt har laddats upp renderas scenen till pixlar. Detta sätt att rendera gör det möjligt att rita samma objekt flera gånger i en scen, med olika parametrar för varje instans. Ett användningsområde för detta är att rendera objekt från olika perspektiv i samma bild, vilket kan utnyttjas för att skapa portaleffekten. Figur 2.1 visar en enkel exempelscen med två kameror.



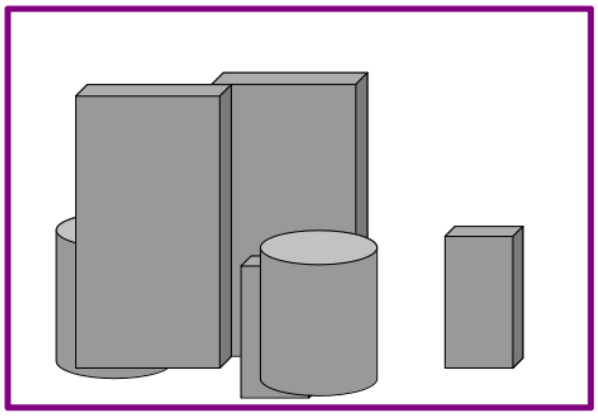
Figur 2.1: En enkel scen med två kameror.

Figur 2.2 visar respektive kameras vy.



Figur 2.2: Bilderna som produceras av den blåa respektive röda kameran i figur 2.1. Till vänster är bilden renderad av den blåa kameran. Till höger är bilden renderad av den röda kameran.

Figur 2.3 visar hur det skulle se ut om båda perspektiven renderades till samma bild. Denna bildkomposition är stökig och svårtolkad, men med ytterligare buffermanulipation är det grunden till en portaleffekt.



Figur 2.3: De två bilderna från figure 2.2 kombinerade.

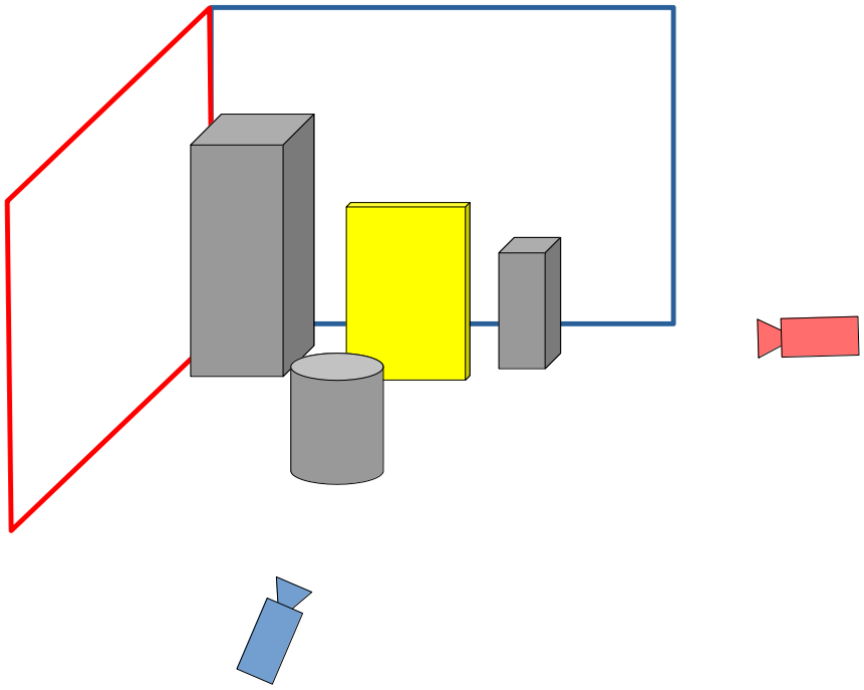
2.2 Rendera portal med stencilbuffer

Stencilbuffern har en byte per pixel som kan sättas när objekt renderars. När scenen sedan renderas kan stencilbuffern användas för att bestämma om en pixel på skärmen ska skrivas till eller ej. Stencilbuffern är initialiserad till 0. Nedan är ett kodexempel på hur en mask av en portalöppning kan skapas med stencilbuffern. Tekniken är baserad på kod av Ingemar Ragnemalm [2].

```
// Aktivera stencilbuffern.
glEnable(GL_STENCIL_TEST);

//  Skriv ej till färg- eller djupbuffern.
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);

// Byt ut värdet i stencilbuffern mot det nya.
```

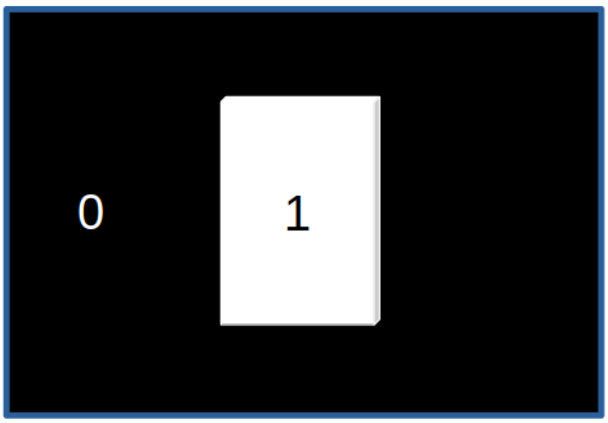
```
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);

// Skriv '1' i buffern där objektet ritas.
glStencilFunc(GL_ALWAYS, 1, 0xFFFFFFFF);

// Rendera portalöppning
renderSceneObject(portalSceneObject, playerCamera);
```

Scenen från figur 2.1 kan byggas ut med ett portalöppningsobjekt, vilket kan ses i figur 2.4.



Figur 2.4: Samma scen som i figur 2.1, men med en portalöppning(den gula, tillplattade kuben).

När koden ovan körs skapas en mask av portalöppningen. Stencilbuffern ser då ut som i figur 2.5. Stencileringen görs ifrån den blåa kamerans perspektiv. Värdet på stencilbuffern blir 1 där öppningen är, och oförändrat 0 annars.

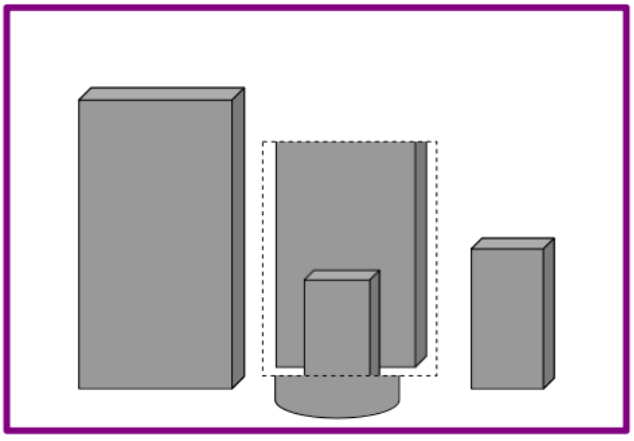


Figur 2.5: Stencilbuffern efter att den gula portalöppningen har renderats med blåa kameran.

Låt målet vara att portalöppningen ska visa den röda kamerans vy. Då renderas först röda kamerans vy där stencilbuffern är lika med 1, och sedan den blåa kamerans vy där stencilbuffern är lika med 0.

```
// Aktivera djup- och färgbuffrarna.
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);

// Behåll stencilen som den är. Vi vill inte ändra på stencilens värden.
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

// Skriv endast till färg- och djupbuffern där stencilbuffern
// är lika med '1'.
glStencilFunc(GL_EQUAL, 1, 0xFFFFFFFF);

// Rendera scenen med RÖDA kameran.
renderScene(scen, rödKamera);

// Skriv endast till färg- och djupbuffern där stencilbuffern
// är lika med '0'.
glStencilFunc(GL_EQUAL, 0, 0xFFFFFFFF);

// Rendera scenen med BLÅA kameran.
renderScene(scen, blåKamera);
```

Figur 2.6 visar resultatet av ovanstående kod. Den streckade rutan markerar vart stencilbuffern går från 0 till 1.



Figur 2.6: Scenen från figur 2.1 från blåa kamerans vy, med röda kamerans vy där stencilbuffern i figur 2.5 är lika med 1.

2.3 Blockera portal med djupbuffer

När stencilbuffern används som i sektion 2.2 uppstår det problem med portalens djup i bilden. Stencilen har ingen information om vilka objekt som ligger bakom eller framför den, vilket skapar en effekt som mer liknar ett hål i skärmen än en portal i scenen. Figur 2.6 visar hur portalen ligger framför cylindern, när den enligt figur 2.4 ska ligga bakom den. Detta problem kan lösas genom att manipulera

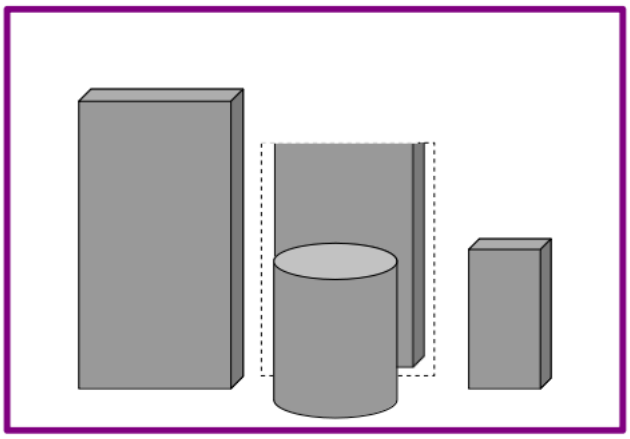
djupbuffern. Tekniken är baserad på lösningen som presenteras i *OpenGL Programming/Mini-Portal* [1].

När objekt renderas i OpenGL sparas också deras djup i bilden. Denna djupbuffer används för att bestämma vilka trianglar av alla objekt som ligger främst i bilden. De trianglar som inte ligger främst kastas bort. Det är dock möjligt att rendera enbart till djupbuffern utan att spara någon färg. Detta gör så att trianglar som tidigare har renderats kommer närmare kameran i djupbuffern utan att ändra deras färg eller utseende. När nya objekt renderas kastas dess trianglar bort om de ligger bakom de gamla trianglarna, men behålls om de ligger framför.

Efter att scenen har renderats genom portalöppningsstencilen från röda kamerans perspektiv stängs stencilbuffern av. Portalöppningen renderas sedan till endast djupbuffern vilket ger trianglarna som syns genom portalen samma djup som portalöppningen. Efter det så renderas scenen utan stencil till färg- och djupbuffrarna med den blåa kameran. De trianglar som ligger bakom portalöppningen kastas bort och de som ligger framför den behålls. Nedanstående kod är ett exempel på hur en portal kan blockeras med djupbuffern på det här viset.

```
// Aktivera djup- och färgbuffrarna.
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);

// Behåll stencilen som den är. Vi vill inte ändra på stencilens värden.
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

// Skriv endast till färg- och djupbuffern där stencilbuffern
// är lika med '1'.
glStencilFunc(GL_EQUAL, 1, 0xFFFFFFFF);

// Rendera scenen med RÖDA kameran.
renderScene(scen, rödKamera);

// Avaktivera färgbuffern.
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// Avaktivera stencilbuffern.
glDisable(GL_STENCIL_TEST);

// Aktivera djupbuffern.
glDepthMask(GL_TRUE);

// Rendera portalöppningen till djupbuffern med blåa kameran.
renderSceneObject(portal1->surfaceSceneObject, playerCamera);

// Aktivera färgbuffern.
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

// Rendera scenen med blåa kameran.
renderScene(scene1Root, playerCamera);
```

Figur 2.7 visar hur bilden skulle se ut när ovanstående kod används.

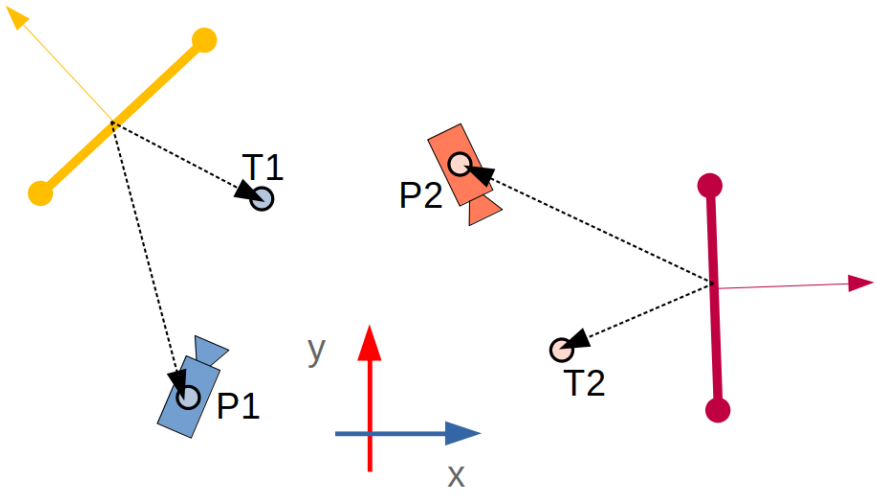


Figur 2.7: En scen med en portal som har korrekt position.

2.4 Beräkna relativt perspektiv

För att få en korrekt effekt av en portal måste perspektivet in i portalen och ut ur portalen vara detsamma. Kamerorna måste ha samma position och riktning relativt till deras respektive portalöppningar. Figur 2.8 visar en blå kamera som ser en gul portalöppning. Den gula portalöppningen(den till vänster i figuren) är kopplad till den lila portalöppningen(den till höger i figuren) som renderas av den röda kameran. En kamera består av kamerans position samt en målposition, vilket är en godtycklig punkt som kameran är riktad mot.

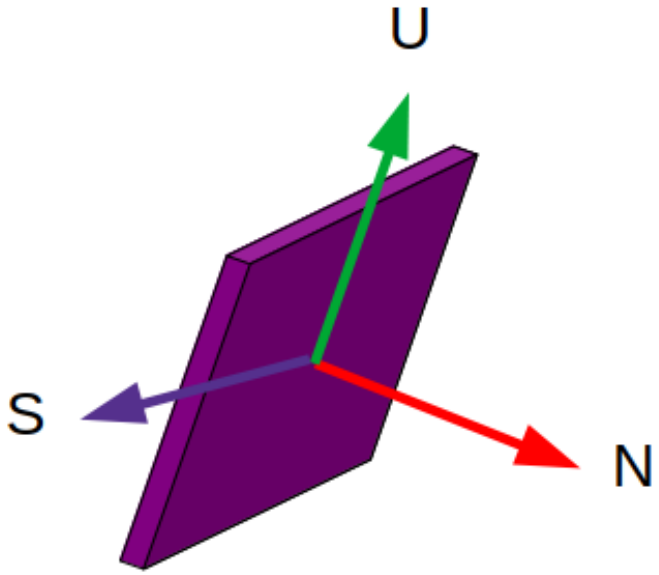
```
struct Camera
{
    // Kamerans position.
    // Ges i globala koordinater.
    vec3 position;

    // Kamerans målposition, det vill säga en godtycklig punkt
    // på linjen kameran är riktad i.
    // Ges i globala koordinater.
    vec3 target;
}
```



Figur 2.8: Exempel på hur portalöppningars kameror translateras, sett ovanifrån.

För att räkna ut positionerna för den röda kameran i figur 2.8 måste båda portalöppningarnas lokala koordinatsystem först bestämmas. Om en portalöppnings position p och normal n är given kan dess koordinatsystem beräknas enligt (2.1). I detta fall antas att portalen inte kan roteras kring normalaxeln. Figur 2.9 visar koordinataxlarna för en godtycklig portalöppning. Koordinatsystemet har sitt origo i portalöppningens position med en normalvektor N som är lika med portalöppningens normal, en sidovektor S som pekar åt sidan, och en uppvektor U som pekar uppåt. Dessa koordinataxlar är givna i världskoordinater. De är ortogonala mot varandra och är normerade.

$$\begin{aligned} \hat{N} &= \frac{n}{||n||} \\ S &= \begin{bmatrix} n_x \\ 0 \\ n_z \end{bmatrix} \\ \hat{S} &= \frac{S}{||S||} \\ \hat{U} &= \hat{S} \times \hat{N} \end{aligned} \tag{2.1}$$



Figur 2.9: En portalöppning och dess koordinataxlar: normal, upp, och sida.

När de två lokala koordinatsystemen har beräknas konverteras den blå kamerans positioner till lokala koordinater för den gula portalöppningen. Dessa koordinater konverteras sedan till globala koordinater med den lila portalöppningens koordinatsystem. Röda kameran får de resulterande positionerna. Den röda kameran hamnar därmed på samma relativa position som den blå kameran relativt till deras respektive portalöppningar. Nedan är kod för att konvertera en punkt från globala till lokala koordinater, givet ett koordinatsystem och dess origo.

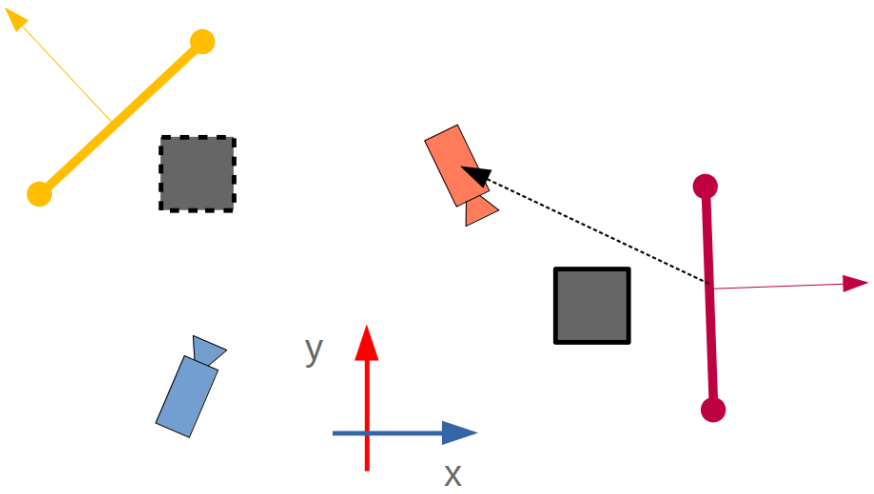
```
// Returns the LOCAL coordinates of a point in WORLD coordinates
vec3 global2local(CoordTriple *system, vec3 origin, vec3 position)
{
    // system   : the ON-axises the local system is based on
    // origin   : the point from which the system has its origo
    // position : the point that is to be converted
    ////////////////////////////////////////////////////////////

    // Reposition to origo
    position = VectorSub(position, origin);
    // How much of each axis does the point go?
    vec3 out;
    out.x = DotProduct(position, system->n_axis);
    out.y = DotProduct(position, system->u_axis);
    out.z = DotProduct(position, system->s_axis);
    return out;
}
```

Nedan är kod för att konvertera en punkt från lokala till globala koordinater, givet ett koordinatsystem och dess origo.

```
// Returns the WORLD coordinates of a point in LOCAL coordinates
vec3 local2global(CoordTriple *system, vec3 origin, vec3 coordinates)
{
    // system   : the ON-axises the local system is based on
    // origin   : the point from which the system has its origo
    // coordinates : the coordinates the returned point has in local
    ////////////////////////////////////////////////////////////////

    // Based at the origin.
    vec3 out = origin;

    // Add coordinates
    // normal = x, up = y, side = z
    out = VectorAdd(out, ScalarMult(system->n_axis, coordinates.x));
    out = VectorAdd(out, ScalarMult(system->u_axis, coordinates.y));
    out = VectorAdd(out, ScalarMult(system->s_axis, coordinates.z));
    return out;
}
```

2.5 Förhindra rendering av objekt bakom portalöppning

Tekniken som presenteras i sektion 2.4 *Beräkna relativt perspektiv* tillför ett problem. I figur 2.10 så renderar den röda kameran vyn den blå kameran ser genom den gula till den lila portalöppningen. För att få korrekt perspektiv så förflyttas den röda kameran bakom den lila portalöppningen. Om ett objekt skulle befinna sig mellan den röda kameran och den lila portalöppningen skulle den också synas framför den gula portalöppningen.



Figur 2.10: Om ett objekt finns mellan röda kameran och lila portalöppningen, så ser det ut som det finns framför gula portalöppningen.

För att slippa detta fenomen tillförs ett krav i renderingen att objekten måste befinna sig framför portalöppningen för att bli renderade. Koden nedan beskriver hur ett sådant test kan implementeras.

```
// Loop through a scene and render all sceneObjects it contains
void renderSceneFromPortal(SceneObject *scene, Portal *activePortal)
{
    SceneObject *current = scene;

    // Check if there is another SceneObject
    while(current->next != NULL)
    {
        // Advance in list
        current = current->next;

        // Get objects local position
        vec3 localPosition = global2local(activePortal->recorderCoord,
        activePortal->recorderPosition, current->position);

        // Render if object is in front of portalOpening
        if(localPosition.x < 0 )
        {
            // Render SceneObject
            renderSceneObject(current, activePortal->recordingCamera);
        }
    }
}
```

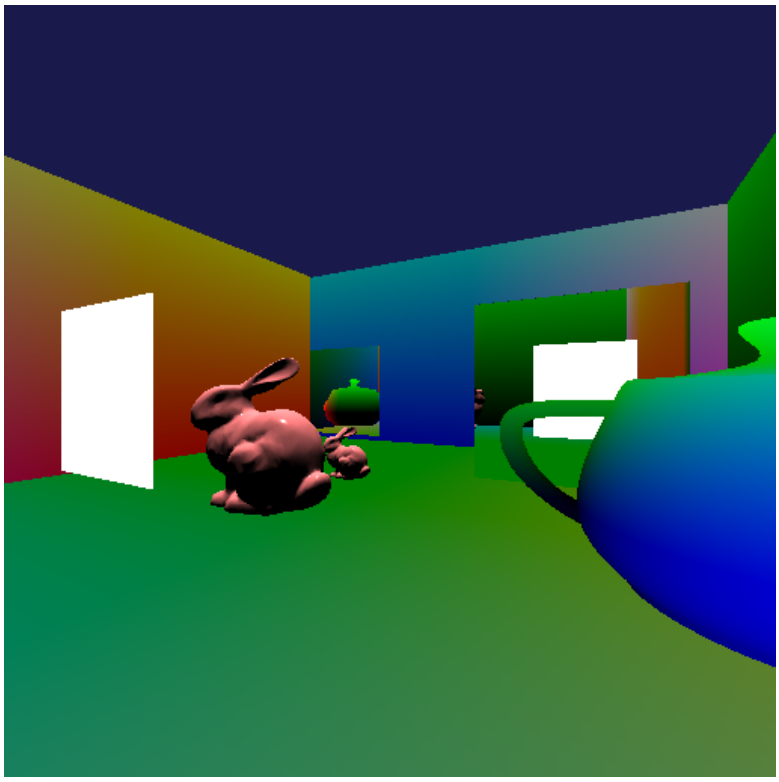
Kapitel 3

Resultat

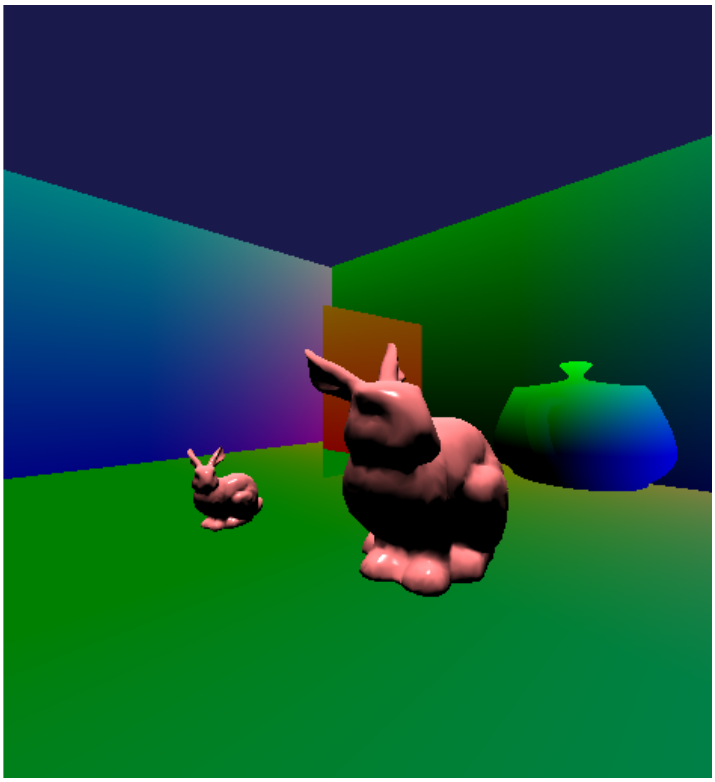
Figur 3.1 visar en exempelscen med en portal i. Scenen är uppbyggd av två rosa kaniner, en tekanna, ett vitt plan, tre väggar, ett golv, samt två portalöppningar.

Figur 3.2 visar att portalöppningen döljs av de objekt som befinner sig framför den, men inte bakom den.

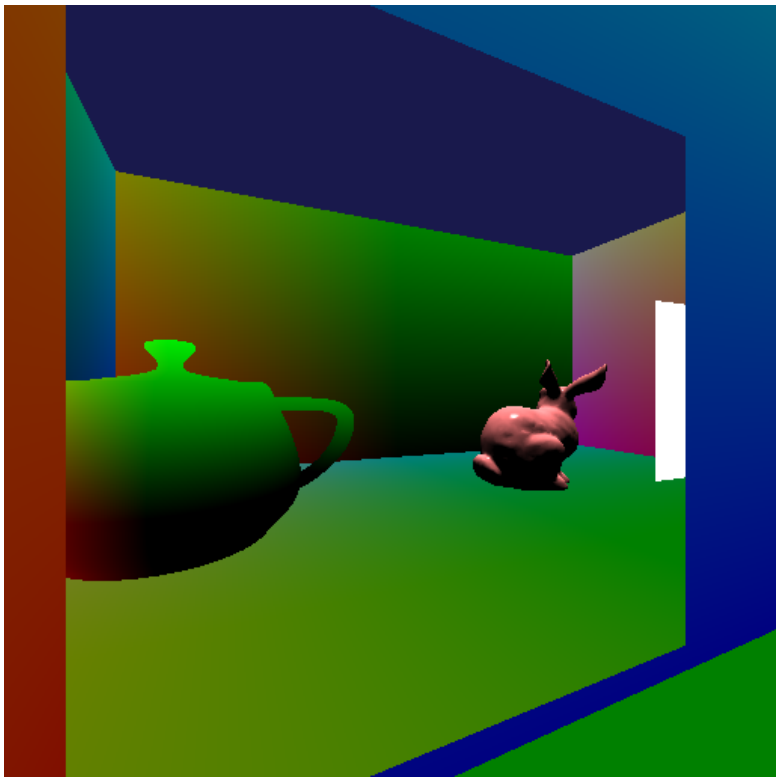
Figur 3.3 visar en närbild på portalen närmast väggen. Upplösningen av portalen skiljer sig inte från övriga scenen och vyn har korrekt perspektiv.



Figur 3.1: En överblick av en scen med en portal.



Figur 3.2: Väggarna är bakom portalöppningen, kaninen är framför den.



Figur 3.3: Närbild av portalen vid väggen.

Kapitel 4

Problem

Vid implementation stöttes en del problem på. Ett första försök med att rendera portaler med *Frame Buffer Objects (FBO)* gav upphov till dålig upplösning och perspektiv. Teleportering hade problemet att det blippade till när kameran gick genom portalen.

4.1 Portal med FBO

Den första tekniken som testades för att skapa en portaleffekt var att rendera en kameravy till en FBO och använda denna FBO som textur på en portalöppning. Tekniken var intuitiv i teorin, men var svår att realisera. Till att börja med var det svårt att få korrekt perspektiv på portalen när den projicerades på en yta som inte var riktad mot kameran. Sedan var det också problem med upplösningen på portalen. Renderingen till FBO använde samma mängd pixlar som slutrenderingen av scenen, så när kameran var väldigt nära portalöppningen var en begränsad del av texturen uppspänd över hela skärmen. Detta gjorde så att pixlarna i texturen kunde ses och gav snarare en effekt av en övervakningskamera än en portal.

Idén om att konstruera portaler med FBO övergavs därför och ersattes med stenciltekniken, vilket inte hade några av dessa problem.

4.2 Blipp vid portalövergång

Vid passage genom portalen är det i en bildruta möjligt att se scenen bakom portalöppningen. Detta är ett känt problem vid rendering av portaler och i *OpenGL Programming/Mini-Portal* [1] ges förklaringen att detta beror på att portalöppningen befinner sig närmre än *z-near* i kamerans frustum. Problemet kvarstod dock i detta projekt efter att det testades att lägga den visuella portalöppningen längre bort än triggerzonen för teleportering, så portalöppningens position är inte den enda förklaringen till varför detta problem uppstår.

För att lösa problemet så kastas den bildruta där teleporteringen sker bort. Att hoppa över den bildruta som är problematisk ger en jämn övergång vid teleportering. Denna teknik kan dock bli synlig vid lägre bildhastigheter. Det skulle vara bättre att åtgärda problemet mer i grunden, men det var inte möjligt inom tidsramen för detta projekt.

Kapitel 5

Slutsats

Portalerna uppfyller kraven som ställdes på dem:

- Varje portal har två portalöppningar som går till varandra.
- Det är möjligt att se in genom den ena portalöppningen och ut genom den andra.
- Vyn genom portalen har samma perspektiv som kameran ser på portalöppningen med.
- Det är inte någon skillnad i upplösningen mellan portalen och övriga världen.
- När en kamera går genom den ena portalen transporteras den sömlöst till den andra.
- Det är möjligt att se båda portalöppningarna för en portal i samma bild.
- Objekt bakom en portalöppning kan inte synas i den andra portalöppningen.

Kravet "*När en kamera går genom den ena portalen ska de sömlöst transporteras till den andra*"är dock inte uppfyllt i extremfall. Vid teleportering så hoppas en bildruta över, vilket kan synas för ett tränat öga eller vid låg bildhastighet. Vid en mer tillämpad implementation skulle denna teknik behöva bytas ut.

Litteraturförteckning

[1] *OpenGL Programming/Mini-Portal*, hämtad: 2019-12-12
https://en.wikibooks.org/wiki/OpenGL_Programming/Mini-Portal##Clipping_the_portal_scene_-_stencil_buffer

[2] Ingemar Ragnemalm *Teknik för avancerade datorspel* hämtad: 2019-12-12
http://computer-graphics.se/TSBK03/

[3] Valve, *Portal* , 2007-10-10, hämtad: 2019-12-13
https://store.steampowered.com/app/400/Portal/