FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Advanced Modifications of a Basic Rendering Framework

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Gernot Winkler
Matrikelnummer 0929255

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 24.10.2014

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Gernot Winkler
Neunkirchnerstraße 17, 2732 Willendorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

Special thanks to Bernhard Steiner for his feedback and support.

Kurzfassung

In diesem Dokument wird ein Überlick über meine Arbeit am Framework der Lehrveranstaltung “Einführung in die Computergraphik” gegeben. Zuerst wird die aktuelle Situation erklärt and welche Erweiterungen gemacht wurden. Es wird beschrieben wie man Aufgaben mit einem ANT-Skript variieren kann. Außerdem werden einige Computergraphik Algorithmen evaluiert, dabei wird erklärt wie diese implementiert wurden oder warum sie nicht verwendet werden und welche Probleme es dabei gibt. Die großen Kapitel dabei sind der Weiler-Atherton Clipping Algorithmus und warum er problematisch ist, außerdem werden verschiedene Methoden für Backface-Culling behandelt und die beiden Beleuchtungsmodelle Oren-Nayar und Cook-Torrance präsentiert. Das letzte Kapitel umfasst eine detailierte Erklärung zum neuen Transparenz Feature welches implementiert wurde und eine genau Erklärung warum “Order Independent Transparency (OIT)” anstatt BSP-Trees verwendet wird inklusive eines umfangreichen Vergleiches.

Abstract

In this thesis an overview about my work on the framework for the lab course “Introduction to computer graphics” will be given. At first the current situation is explained and what extensions should be made. It is described how task variations are implemented with an ANT-script. Also some computer graphics algorithms are evaluated on how well they would perform in the framework and it is explained how they are implemented or why not. The major topics are Weiler-Atherton clipping and why it is problematic, different methods for backface culling and the Oren-Nayar and Cook-Torrance shading models. The last chapter explains the new transparency feature that has been implemented in detail and why order independent transparency (OIT) has been chosen over BSP-Trees including a extensive comparisons.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The course "Einführung in die Computergraphic" (introduction to computer graphics), formerly called "Computergrafik 1" (computer graphics 1) uses a java based software rendering framework [4] to teach students a modern graphic pipeline. The main render methods are blank for the students and are implemented during the course. The main reason for this thesis is that the tasks for students were the same for some semesters now and copying the code from colleagues is easy to do. Plagiarism checks don't work well since there is only little difference between correct solutions. The reference solution is also available to students after each task, so every student could have had access to last year's reference solution. This reference solution is given to the students so they can work with the reference code for the previous task so errors don't have effects later on. The goals of my work were to implement a system that allows to individualize the tasks for each student and to experiment with new algorithms that could work instead the currently used ones.

1.2 Previous Work

As stated above the framework is a java based software renderer. The most important features are loading/saving collada models/scenes and render models with different shaders. Other features are movement, rotation and translation of camera, models and lights. Point lights and directional lights are available. It is also possible to toggle backface culling on/off and to switch between line and fill rasterization. A lot of these features are implemented by the students during the course. A screenshot of the framework is shown in figure 1.1.

Up to now the course has six tasks for the students to implement.

Task 1

The first introductory task is to implement Bresenham's line rasterization algorithm.

Task 2
The second task are basic mathematical methods, like vector and matrix multiplication.

Task 3
This task's goals are color interpolation, and viewing projection. Also clipping is implemented with the Sutherland-Hodgman algorithm in an iterative form.

Task 4
In this task the students use fixed precision variables, implement z-buffer depth checks, barycentric fill rasterization and backface culling with signed triangle area method.

Task 5
Task five is all about shading. Students implement the Lambert and the Blinn illumination model and Phong/Gouraud shading. Resulting in following four shaders: LambertGouraud-Shader, BlinnGouraud-Shader, LambertPhong-Shader, BlinnPhong-Shader

Task 6
The final task is to implement perspective correct interpolation and texturing. Also a custom shader should be implemented by students where they can be creative and implement some more complex and good looking shaders.

Renderloop architecture

The start point for frame rendering is by calling scene.draw() from the render loop. The scene class contains all information for objects in the scene:

- Objects
- Lights
- Cameras
- Geometries

The scene.draw method (see algorithm 1.1) gets a Renderer and a Framebuffer as arguments. The Renderer class handles rendering of a geometry and the Framebuffer handles writing of pixels to the current frame.
In the renderer.draw method (see algorithm 1.2) a mesh runs through the whole rendering pipeline.

The rasterizer now draws the vertices to the framebuffer. The Rasterizer class itself is an interface to allow switching between line rasterization and fill rasterization and after my changes also to switch the algorithm, like DDA and Bresenham for lines or barycentric and scanline for fill rasterization.

```
//scene.draw(Renderer,Framebuffer);
get viewing- and projection matrix from active camera;
foreach geometry do
    foreach surface (geometry.getSurfaces()) do
        get shader and mesh from surface;
        set parameters (MVP matrix, lights,camera,model- and normal matrix);
        call Renderer with render.draw(shader,mesh, framebuffer);
    end
end
```

Algorithm 1.1: scene.draw() pseudocode

```
//renderer.draw(shader,mesh, framebuffer);
loop over vertices and call vertex shaders (Transformation from Mesh.Vertex to Vertex);
build primitives (triangles) and call the clipper (polygons are no longer triangles after
  clipping);
transform vertices to screen space;
do backface culling;
call rasterizer;
```

Algorithm 1.2: renderer.draw() pseudocode

The build script

A very important part of the whole application is the powerful buildscript. It is written in ANT[1] and can build all student packages for each task, including reference content and a build script for the students to generate a submission package. It can run every specific task and also generate webstarts for the course staff for checking a student's submission. The generated student packages contain an obfuscated framework with only those classes as source files students need for the implementation or are important for understanding the architecture of the framework. During the build process a python script is used to switch between reference code and empty code parts to fill out, depending on which task should be build. This code switching works with text macros as comments in the source file.

1.3 Goals and Contributions

The specific goals of my work were to implement a system allowing to switch tasks and experiment with different algorithms that could be used instead of the current ones.

Task 1

Up to now Bresenhams algorithm is used for line rasterization. In the repository an old code for the Digital Differential Analyzer (DDA) [13] algorithm existed that was used

[1]http://ant.apache.org/

some time ago. This code was updated to work in the current version and can be switched as alternative task 1.

Another algorithm I tried was Xiaolin Wu's line algorithm [15]. This algorithm does line anti-aliasing, but in the first task the framework can't handle transparency, which would be required to make use of it, so it would not fit as introductory task and the idea was discarded.

Task 3

In the third task clipping is implemented with Sutherland Hodgman's Algorithm [10] in an iterative way. The repository contained an old code for a recursive implementation, this code was fixed for the current version and made possible to use as alternative for task 3.

As new possible variation Weiler-Atherton's algorithm [14] was evaluated. This was very extensive and a whole chapter is dedicated to it. The final decision was not to use it, which is detailedly explained in Chapter 3.

Task 4

In task 4 fill rasterization is implemented, currently with a barycentric triangle rasterizer, but old code for a scan-line based fill rasterizer existed, again this code was updated and made possible to use as alternative to task 4.

As new algorithm Warnock-Subdivision [12] was evaluated, but this algorithm would not really fit in the frameworks current render-loop so this idea was discarded early.

Also backface culling is implemented in task 4, the current way is by calculating the signed triangle area, as alternative the dot product between eye and normal was implemented.

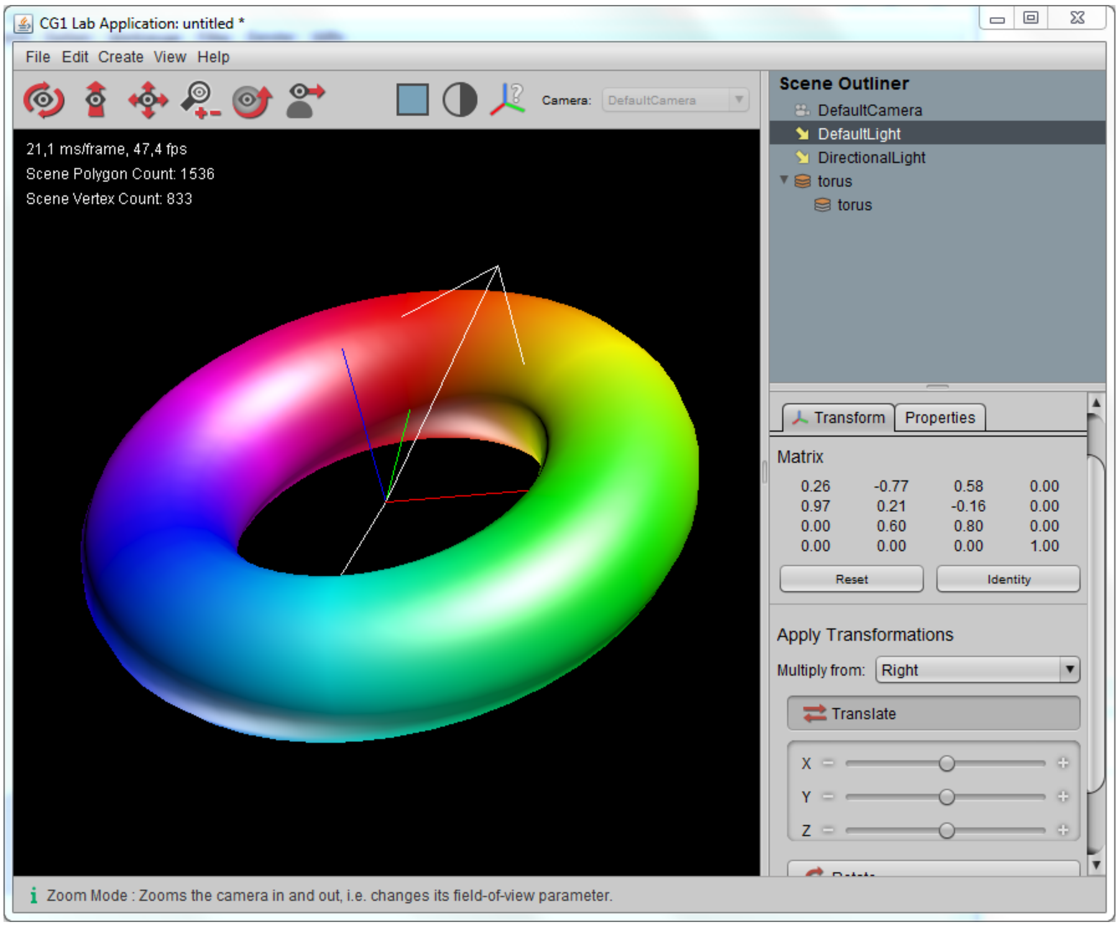
Task 5

In this task Lambert-Gouraud shading, Blinn-Gouraud shading, Lambert-Phong shading and Blinn-Phong shading are implemented. For this task I made reference implementations for two more complex shading models Oren-Nayar [8] and Cook-Torrance [3].

new Task

As completely new task alpha blending was added to the framework. Two different methods, back-to-front drawing with BSP-Trees [5] and A-Buffer [2], were implemented and tested. The A-Buffer implementation seemed superior and has been added as new possible task. This is fully explained in Chapter 6 of this work.

Another idea that was discarded early on was a stencil buffer. The framework does not support multiple render passes, thus it would not really be possible to make use of it. An architectural change would be required for that.



21,1 ms/frame, 47,4 fps
Scene Polygon Count: 1536
Scene Vertex Count: 833

Camera: DefaultCamera

Matrix

```
0.26   -0.77    0.58   0.00
0.97    0.21   -0.16   0.00
0.00    0.60    0.80   0.00
0.00    0.00    0.00   1.00
```

Multiply from: Right

Zoom Mode : Zooms the camera in and out, i.e. changes its field-of-view parameter.

Figure 1.1: A screenshot of the framework

CHAPTER 2

Switching Tasks with ANT

2.1 Goals

The goal for task variation is to keep the code as clean as possible and don't leave traces of other tasks for students. This means that a student should only see "TODO" comments in his/her code he/she actually has to implement. It should also be easy for the staff to change which variation is used and it should be possible for them to add new task variations without too much effort. I wanted to use technologies that are already used in the framework and don't add new dependencies, as the whole application should work in newer versions of java and ANT without problems.

2.2 Technical background

Apache ANT[1] is a build management tool written in java by the Apache Software Foundation. Build scripts themselves are XML files. The main use for ANT is to automatize the whole building process, but it can also be used to run the software or create submission packages.

2.3 Implementation

The specific problem that should be solved is a way to exchange classes in the framework, for example student A has to implement line rasterization with Bresenham's algorithm while student B should use the DDA algorithm. A possible solution would have been to replace code with the python script that is used for code changes between tasks, but that would make the source code very unreadable and implementations that require different arguments would be a lot more difficult to implement and cause a very bad maintainability. This lead me to try switching classes with the ANT buildfile. The solution I came up with is a new class called `TaskManager`, that

[1]http://ant.apache.org/

returns the correct instance of a class for a specific task. There was already an `Rasterizer` interface that both, the line and fill rasterizers, implement. For other tasks an interface had to be extracted and in the case of backface culling a method had to be moved to a separate class.

In the example of task 1 there are now two classes `BresenhamRasterizer` and `DDARasterizer` but the application never instantiates them directly. The `TaskManager` has a method for each task variation to return an instance, in this case a `Rasterizer` (an interface implemented by both classes). Which instance is returned is determined by the ant script. During the build process, the ant script does a string replacement in the source file of the TaskManager, by replacing the dummy value of a private variable to the classname of the implementation to use. To avoid modifications of the original source, the directory is copied to "src_copy" before replacement and the application is compiled from there. The `TaskManager` returns instances by calling `class.forName( "Classname as String" )`. This potentially unsafe operation is only called during the build process when creating the packages for the students, so there is no danger that students might do something bad there since this script is only available to the staff.

Now the most important part is how the build script handles this: As already explained the actual class to use gets replaced in the source file, this is achieved with ANT's replace task. The more difficult problem is the selection of the correct class, since ANT does not support arrays or something where one could just specify an index to choose which class gets returned. There is a workaround by creating some internal dummy ant targets that set variables and others that only do something when a variable is set. To change the used variation only one single property has to be changed for each task.

```
<!-- Line Rasterizer: 0 for Bresenham, 1 for DDA-->
<property name="task1.var" value = "0" />
```

Everything else is handled by internal ANT targets depending on this value. The build task needs to be dependent on all these dummy tasks, to keep it clean there is a dummy task that dependents on all others.

```
<!-- This sets all variables for task variations to
 include/exclude the correct files -->
<target name="-setup-task-variations" depends="-task1-setup,
 -task1-use-var0, -task1-use-var1, -task1-use-var2,
 ...other tasks...,
 -taskvar-invalid-selection">
<!--dummy target to call all others in correct order-->
</target>
```

The setup tasks (taskX-setup) work the following:

```
<!-- set conditions for task1 class selection -->
<target name="-task1-setup">
```

```
 <!--Task 1 Line rasterizer to use-->
  <condition property="task1.useVar0" value = "true">
   <equals arg1="${task1.var}" arg2="0"/>
  </condition>
  <condition property="task1.useVar1" value = "true">
   <equals arg1="${task1.var}" arg2="1"/>
  </condition>
  <condition property="taskvar.invalid" value = "true">
  <not>
   <or>
    <equals arg1="${task1.var}" arg2="0"/>
    <equals arg1="${task1.var}" arg2="1"/>
   </or>
  </not>
 </condition>
</target>
```

This sets `task1.useVar1` or `task1.useVar0` depending on the value of `task1.var`. `Taskvar.invalid` is set if ones variation is set to an invalid value to stop the build process. The `taskX-use-varY` tasks now set classnames required for including/excluding. The dummy targets above are required because the ANT if statement can only check if a property is set or unset and the `condition property/` statements can only set one variable. Setting all variables with copied condition checks would be error prone and even more unreadable.

```
<target name="-task1-use-var0" if="${task1.useVar0}">
 <property name="task1.rasterizer" value = "cg1lab.render.
     BresenhamRasterizer"/>
 <property name="task1.exclude1" value = "cg1lab/render/
     DDARasterizer.java"/>
</target>
```

These variables are now used to replace the string variable in the code that contains the classname and to exclude the other file during the build process since they are not needed and the students should not get confused by source files they don't need.

CHAPTER 3

Clipping in Screen Space with Weiler-Atherton

3.1 Clipping in the Framework

As explained in Chapter 1 the clipping happens during the `renderer.draw()` call after vertex-shading and before screen space transformation. The current implementation uses Sutherland-Hodgman's algorithm which can be used in world space. Weiler-Atherton clipping should happen in 2-dimensional space, so instead of clipping before screen space transformation, this clipping will be called directly afterwards.

3.2 Theoretical background

The clipping algorithm from Weiler and Atherton [14] is for clipping a polygon against another one with only the minimal number of polygons as result. These polygons are called the subject polygon and the clip polygon and the algorithm handles clipping the subject polygon against the clip polygon. Both polygons can be concave and can contain holes but the winding is important. Polygons are represented by circular vertex lists in clockwise order while each hole is a circular list in counterclockwise order. The original algorithm as explained in the paper can handle holes, return the outside polygons that get clipped away and the result can also consist of multiple polygons that are not connected. But this is all not required in the framework since there will never be holes and polygons that get clipped away are not needed. It will also never happen that the subject gets cut in half so the result is always exactly one polygon. The algorithm will be explained in a simple form with those parts left out that are not required in this application.

Preparation

The first step is to setup the data structures required for the algorithm to work. At the beginning each polygon is represented by a list of vertices. The original algorithm uses clockwise order but since triangles in the framework are in counterclockwise order it is easier to keep it like that. It is only important that both polygons have the same order. Now it is required to calculate all intersection points between both polygons, which need to be inserted in both polygon's list in correct order. It is necessary that intersection vertices are marked somehow because later on they need to be treated differently than the original vertices. Another important thing is that every intersection point is connected or linked to the other point on the other polygon that represents the same intersection. This is required because in the next step the list of polygons is followed and at intersection points a jump to the other polygon at the same position is made.

The actual clipping

a) Get a starting point by finding the first inbound intersection. That can be achieved by following the vertex list of the subject polygon until an intersection is reached and the vertex before was outside of the clip polygon. This intersection point is the first vertex for the resulting polygon.

b) Follow the subject polygon from vertex to vertex until an intersection point is found. All traversed vertices are part of the resulting polygon.

c) Switch to the other polygon at this intersection point.

d) Follow the clip polygon and add all vertices to the result until an intersection is reached.

e) Switch back to the subject polygon at this intersection point.

f) repeat the procedure from “b” to “e” until the starting point is reached. The resulting polygon is now closed.

3.3 Implementation in the framework

Since this algorithm is for 2D polygons the best way to use this for clipping is in screen space. Every polygon is already projected to two dimensions and converted to screen coordinates at this stage. So the idea is to clip every polygon against the screen rectangle. At this stage in the framework every polygon is a triangle so it's always a rectangle as clip polygon and a triangle as subject.
The framework uses Sutherland-Hodgman [10] algorithm for clipping until now. The main advantages of Weiler-Atherton compared to Sutherland Hodgman are:

- Can clip concave polygons with holes
- Results in minimal number of polygons

- Can return outside polygons too

But in this case none of these advantages is relevant, as already stated it's always clipping a triangle against a rectangle. There is also no advantage in returning only one polygon as result because the next step in the framework is fill rasterization where every polygon is broken down in triangles anyway. Outside polygons are never needed when clipping against the screen rectangle.

The disadvantages:

- More complex implementation
- Needs to store more values (lists, intersection points)
- Not well suited for hardware implementation
- Special case handling

Since the framework is a software renderer it does not matter that Weiler-Atherton clipping is bad for use in hardware because it has to handle multiple lists, but the other disadvantages do apply. So it results in only disadvantages compared to Sutherland-Hodgman in this application. But it still might be useful as alternative task for students for teaching purposes.

3.4 Problems

While the algorithm seems easy in the general case, which it actually is, there are problematic situations that need to be handled with caution.

Problem 1: Vertex lies on others edge

The first case that needs special handling is when an edge of the subject polygon goes through a vertex of the clip polygon, in the case of this application when a triangle edge goes through the screen corner, resulting in an intersection point that is exactly the vertex of the screen rectangle. The normal case in this algorithm is too follow one of the polygons and switch too the other one at intersection points. This does result in wrong polygons when this special case occurs. The easiest solution I found while implementing is to check for this case and not to switch the polygon when this happens. This does solve this issue.

The other possibility is when one vertex of the subject lies on the screen rectangle. This case does not cause problems if there are other intersections, because if only one vertex lies on the edge and both other vertices are outside or both are inside, the triangle does not need to be clipped or can be fully clipped. The problematic case is when two vertices lie on the same edge, this is explained in the next subsection.

Problem 2: Shared edge

When the clip polygon and the subject polygon share an edge, the algorithm does not work since intersection points can't be calculated. The easiest solution to this problem is to define a shared edge as inside or outside and not to clip or fully clip in this case. In this application it's always a triangle, so if two points of the triangle lie on the same edge of the screen rectangle the solution is to look at the third vertex. If this vertex is in the rectangle nothing needs to be clipped, otherwise it's outside and thrown away. There might be a minor artifact because a one pixel line on the side is not visible while it actually should, but while testing I couldn't really notice it. If it would be a problem it still could be worked around by not throwing away the triangle but by replacing it with a line on the shared edge by setting the third vertex to be a duplicate of the second one which results in a triangle that is actually a line. While this is not a problem in this case, if such triangles exist before clipping this leads to the next problem.

Problem 3: Triangles that are lines

At this point the complexity of special case handling already exceeds the normal case. But this special case makes it even worse. When a triangle happens to be a line there are two exact same intersection points at the same location. At first this seems to be a bad model because a triangle should never be a line, but this is actually the way to render lines with fill rasterization and completely valid for the fill rasterizer classes. The framework renders its coordinate gizmo this way. This case makes ordering the vertices in the vertex chain not deterministic because two vertices need to be at the same location. This case would require another special treatment. The best way seems to check for line triangles and treat them completely different. At this point the complexity of special cases exceeds the normal algorithm by far, so the decision was made to not use this algorithm as task for students in an introductory course.

CHAPTER 4

Backface Culling

4.1 Culling in the Framework

Backface culling is removing faces before rendering that face backwards which means they are not visible. Technically a face is a backface if it's normal vector is pointing away from the camera. Triangles in the framework are defined to have a counter clockwise winding. The framework does already include backface culling as a task for students to implement. There are different ways to calculate if a triangle is backfacing, but the framework was hardcoded to only use one method. In the rendering pipeline of the framework backface culling happens after screen space transformation, which is also after clipping. To make it possible to have different culling algorithms the method `isBackfacing` was extracted to a new class implementing a `Culler` interface. This allows switching the culling algorithm with the ANT script and the `TaskManager` class.

4.2 The different algorithms

Signed area of the triangle

The old way in the framework is to check the sign of the triangle area. The cross product of two 3D vectors results in a normal vector with twice the length of the area from the triangle created by those two vectors. For 2D triangles, like they are in screen space, the third coordinate can be assumed as zero. For area calculation the absolute value is used, but by checking the sign it's possible to determine if a triangle is back facing or not. If the area is negative the triangle is looked at from the back side and can be discarded.

Angle between normal and eye vector

Another method was implemented as alternative task for backface culling. This method is to check the angle between the normal of the face and the eye vector. The dot product between two vectors can be used to get the angle between these vectors. By checking if the angle is greater than 90 degrees the triangle can be discarded because it is facing away.

CHAPTER 5

Advanced Shading Models

5.1 Shading in the Framework

The framework already supports shading in a similar way in the modern OpenGL rendering pipeline with vertex and fragment shaders. There is a shader interface that contains `shadeVertex` and `shadeFragment` methods that simulate vertex and fragment shading and an abstract class called `SurfaceShader` that implements the interface and also handles passing arguments like matrices and camera location. All shaders in the framework extend this class. Using annotations additional parameters can be defined that are passed by the user interface. Adding additional shaders is already possible. Currently four shading models are up to be implemented by students. Two Gouraud shaders with Lambert and Blinn illumination and Phong shaders with both illumination models. Early in the course a color shader is implemented and in the final task a texture shader is programmed. A task for students is also to implement a custom shader where they can choose from a lot of different methods and be creative. The framework does support tangent space so normal mapping is possible too.

5.2 Goals

The current shader combinations for students are not very difficult and my task was to create reference implementations for more complex shading models than Blinn-Phong. After a bit of research the decision was that Oren-Nayar [8] and Cook-Torrance [3] would be a good choice.

5.3 Oren-Nayar Shading Model

The Oren-Nayar [8] reflectance model is a microfacet model and the goal is to more accurately simulate the diffuse reflectance of rough surfaces since Lambertian reflectance does not give realistic results for these surfaces. In the Lambertian model, the illumination intensity is calculated by the angle between the surface normal and the light direction vector. This gives realistic results for smooth surfaces but in computer graphics complex surfaces are usually not modeled because of performance reasons and only faked by textures or shaders. Oren-Nayar's reflectance model does simulate such a rough surface quite realistically. A round object like a cylinder does appear very dark at the sides when rendered with Lambert's model, while Oren-Nayar results in an overall more bright result. The brightness from points directly hit by light are equally, but the drop in brightness is less for points with an higher angle. Oren-Nayar's reflectance model is based on the Torrance and Sparrow [11] model. Surfaces are assumed to be a collection of V-cavities where each cavity consists of two planar facets with Lambertian reflectance. The roughness of a surface is calculated with a probability distribution of slope areas. The Gaussian distribution is often used and the standard deviation sigma is used as parameter for the roughness. With a roughness value of zero the calculations simplifies to Lambert's model. A comparison of Oren-Nayar with roughness of 0.5 and Lambert-Phong in the framework is shown in Figue 5.1

Implementation

For the java implementation my main source was a HLSL implementation [7]. The original equations are quite performance heavy compared to simpler models since six trigonometric functions have to be called. Simplifications with minimal effect are already proposed by Oren and Nayar. Subterms with only minimal effect can be left out. For GPU purposes it is also good to change all spherical coordinates and trigonometric functions to vectors when possible. Since the framework tries to mimic a GPU in shading I did follow these steps in my reference implementation. Only the roughness is required as additional parameter compared to simpler models like Blinn-Phong.

5.4 Cook-Torrance Shading Model

Like Oren-Nayar's reflectance model, the Cook-Torrance [3] model is also a microfacet model. While Oren-Nayar only handles diffuse lighting and is targeted at matte surfaces, Cook-Torrance does simulate specular highlighting and was invented for more shiny materials like metals and plastic. Compared to Phong reflection, Cook Torrance gives more realistic results. The main difference lies in the more complex specular lighting calculation. Like Oren-Nayar, a surface is considered rough, but all facets are mirrors and have specular reflection. This results in a lot more specular reflection outside of the main specular highlighted spot like in Phong or Blinn-Phong. The microfacets are considered very tiny, smaller than what can be displayed, and the model handles effects like self shadowing and masking. Shadowing is when incoming light is blocked by facets and masking handles blocked reflected light.

Implementation

Again I used gpwiki.org's HLSL reference [6] as main source for my implementation. The original formula is complex and performance heavy so again simplifications can be done. The specular calculation consists of a geometric term, a roughness term and a Fresnel term. The geometric part consists of vector calculations which is not really a difficulty. The roughness term can be calculated with Beckmann's distribution [1] which gives quite accurate results for a lot of materials but is a bit performance heavy. A more efficient solution is a Gaussian distribution but this requires a second parameter which has to be set for each material, while Beckmann's distribution only needs the roughness as parameter. For the framework I chose Beckmann over Gauss mainly because of one parameter less and pure frames per second is not more important than good looking pictures. For real-time applications when performance matters a lot, a Gaussian distribution might be more feasible. For the Fresnel term, which is responsible for non constant color of reflected light, Schlicks' approximation [9] can be used since the original equations are way too complex. As parameter this approximation requires the Fresnel reflectance at normal incidence which results in two additional parameters in total for this shader compared to Blinn-Phong. Figure 5.2 shows the Cook-Torrance model with high reflection and roughness values compared to Blinn-Phong shading.

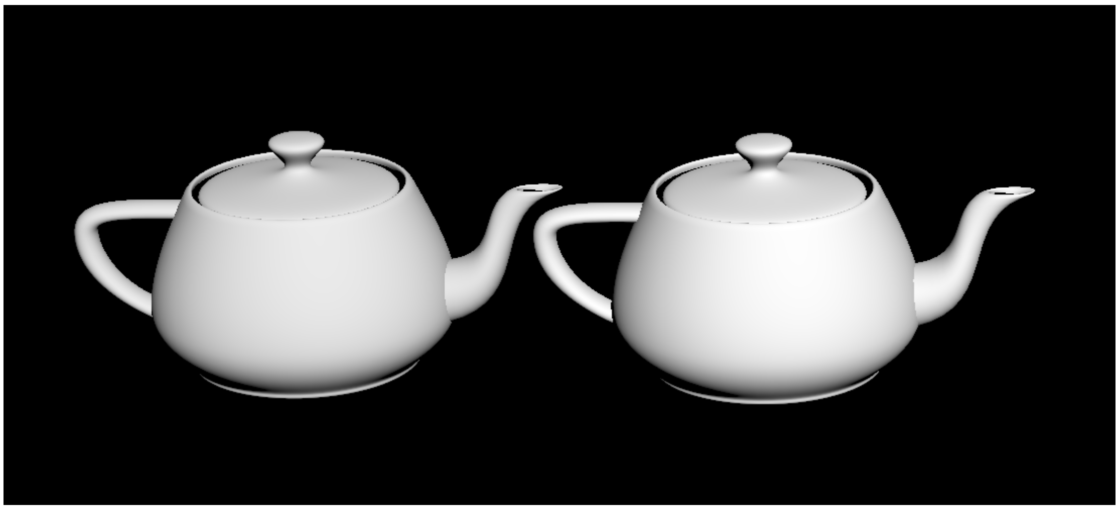


Figure 5.1: Left: Oren-Nayar, Right: Lambert-Phong

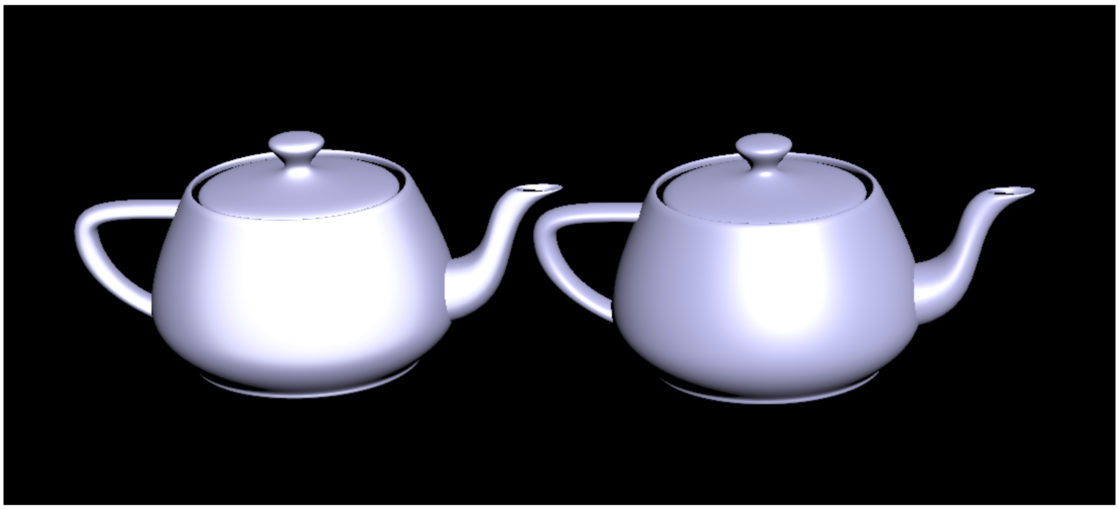


Figure 5.2: Left: Cook-Torrance, Right: Blinn-Phong

CHAPTER 6

Alpha Blending

6.1 Current Situation in the Framework

Up to now, the framework does not handle transparency in any way. The program tries to mimic a modern OpenGL pipeline with vertex and fragment shaders, but alpha values are not used. During the drawing process color values are stored in the framebuffer as an array of RGB values converted to Integers. In addition, depth information is stored in a Z-Buffer. Fragment shaders return three-dimensional vectors as color values and those are stored in the framebuffer if they pass the depth test (no fragment closer to the camera is already at this position). Scene objects do not have a deterministic order in the framework, since they are stored in hash maps which are unordered collections.
Alpha Blending is a completely new feature in the framework. My goals were to experiment with different methods and create a working reference implementation that can be used as base for a task for students. The two different techniques I tried are back to front drawing using BSP-Trees and A-Buffer, which is technically a list of color and depth value pairs for each pixel in the framebuffer in this application, also referred to as order-independent transparency (OIT).

6.2 Basic Transparency Handling

No matter which method is used for sorting/ordering, some basic changes to the framework are needed. First, shaders need to return a four-dimensional vector because without an alpha value storing opacity, there can't be transparency. The `shadeVertex` method of the `Shader`-interface, which is implemented by all shaders, now has to return a `Vec4`. This is not a heavy change, since all shaders now only need to additionally add an alpha component to their previous `Vec3` return value. The alpha value is completely independent from everything else and is passed as parameter from the GUI. This architecture also allows to do more complex things with alpha. It is, for example, possible to return a different 8bit alpha value for each fragment. Alpha values could be loaded from textures or changed depending on viewing angle.

The other change that both methods need, is a blending function in the framebuffer that combines two colors depending on their alpha values. A simple A over B alpha compositing would be enough, but to be closer to OpenGL the blend function tries to mimic the behaviour of `glBlendFunc` which has a source and a destination factor as parameters so different blending techniques can be used. In the framework, this setting was added to the GUI to globally set the blend function behavior. This allows to correctly blend two colors together, but for rendering a whole scene this is not sufficient. Blending A over B does not give the same result as blending B over A. To get correct results, the so called painter's algorithm is used which simply orders polygons and draws them back to front. While the concept of drawing back to front is trivial, sorting isn't. Even sorting per polygon is not sufficient because there are cases with intersecting polygons, where none of the polygons is completely before or behind the other one. To solve this problem, more complex methods are necessary. Two possible solutions are BSP-Trees or per-pixel lists in the framebuffer, which is called order-independent transparency.

6.3 BSP-Trees

One solution to solve problematic cases is by removing them. BSP stands for Binary Space Partitioning where the goal is to create a binary tree for a model to get correct ordering for all faces. Unsolvable cases are subdivided and turned into trivial ones.

Theoretical Background

A BSP-Tree [5] is a binary tree of polygons that subdivides a mesh along hyperplanes. This allows to have a deterministic ordering of polygons because problematic cases are eliminated. In a BSP-Tree the root node contains a polygon and a hyperplane spanned by this polygon. The subtrees contain all polygons in the positive and negative subspace split by this hyperplane. To create a BSP-Tree, an arbitrary polygon is chosen as starting point. There is room for optimization by choosing a polygon that creates a balanced tree but for functionality this does not matter. Every polygon lies on a plane in the three-dimensional room. This plane splits the room in a negative and a positive half space. The triangle equation is used to calculate in which halfspace a polygon lies. All other polygons except the root polygon are now put in either the left or the right subtree, depending whether they are completely before or completely behind the plane the root nodes' polygon spans. All polygons that are not completely in one of the halfspaces intersect this plane or lie exactly on it. Intersecting polygons must be clipped against this plane and one half is put in the right and the other one in the left subtree. All polygons that lie exactly in the same plane can be stored in a list in the root node. This whole procedure is repeated recursively for both subtrees until there is only one polygon left.
To render a BSP tree, it is traversed depending on the position of the camera. The algorithm is pretty easy: If the current node is a leaf node it is rendered. In case of interior nodes, if the camera location is before the plane of the current node, the negative subspace (=faces behind) is rendered before the current node and the positive subspace is rendered last. When the camera is behind the splitting plane, the order is positive subspace before the current node and the negative subspace is rendered last.

BSP-Tree Limitations

A BSP-Tree for a model can only solve inner object ordering. When multiple objects intersect each other, it's necessary to combine BSP-Trees together to create one big BSP-Tree for all the objects combined. While a BSP-Tree for a single object only needs to be created once and does not change as long as there are only affine transformations applied to it, the combined BSP-Tree unfortunately must be recreated as soon as a single part of the BSP tree receives a different transformation. This means the combined BSP-Tree has to be rebuilt every time a scene object receives a change to it's model matrix. This is way to performance heavy for most applications. But as long as objects do not intersect each other, which often is the case in a scene, it is sufficient to order all objects from back to front and use BSP-Trees just for inner object ordering. This does give correct results if no intersections between objects are present. In Figure 6.1 an example is shown where ordering can't be solved without combining trees, the torus is both before and behind the cube which is not solvable with this implementation. Another restriction is that BSP trees do change the model which is visible when using wireframe rasterization and that may look a bit odd.
For an implementation in the framework, the solution with ordering objects and using BSP-Trees for inner object order seems feasible.

6.4 Order Independent Transparency (OIT)

Another solution to avoid such ordering problems is to use techniques where the order of objects do not matter for transparency.

A-Buffer Adaption

A-Buffer [2] stands for anti-aliased, area-averaged, accumulation buffer and is used for resolving visibility and aliasing between intersecting opaque or transparent objects. The original purpose was anti-aliasing but this method can also be used for transparency. By adapting the idea of saving a per-pixel list of fragment data, it is possible to solve the ordering problem in the framework to get correct results that do not depend on the order in which objects are passed.
In the framework, each time the `setPixel` method is called a depth check is performed and if it passes, the color value is stored in the framebuffer at this position. To extend the functionality to work with transparency a new `aBuffer` array with the same dimensions as the framebuffer is added. This new data structure contains a list for each pixel, where each element of the list consists of color, alpha and depth. In the `setPixel` method now it is checked if the color value is transparent (alpha is less than one) and if so the value is added to the A-Buffers list instead of written to the framebuffer. Opaque values are still handled like before. When all objects are drawn, the `blendABuffer` method is called. In this method every list is sorted back to front and then blended into the framebuffer. Every element behind the current framebuffer depth is discarded, all others are blended over the current value and written to the framebuffer. Since these lists do not interfere in any way, it's possible to speed this up with threading. This gives correct solutions except when two polygons are exactly (or nearly but with smaller difference than Z-Buffer resolution) on each other which is not solvable anyway.

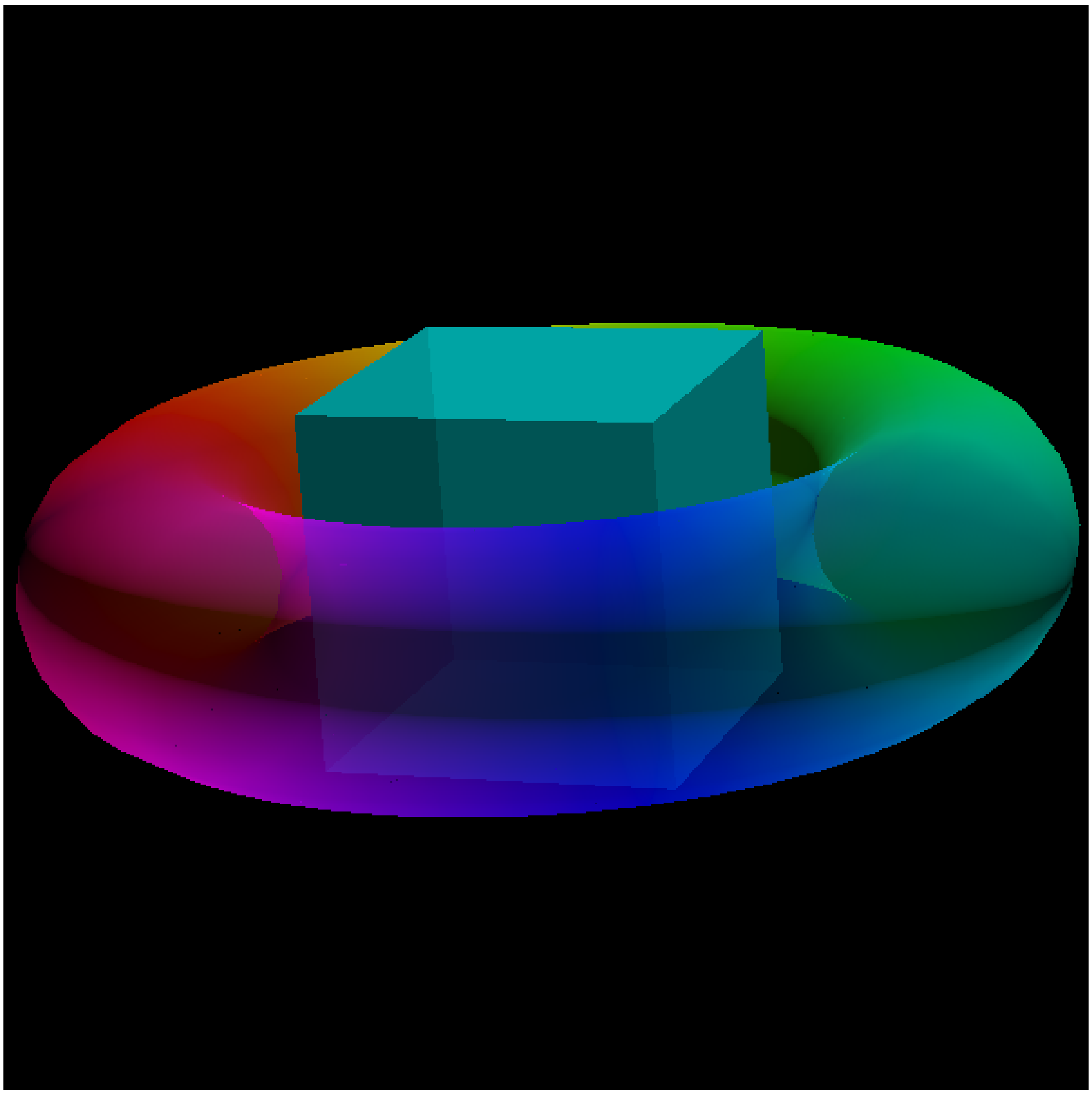


Figure 6.1: Intersecting transparent objects with BSP-Tree implementation

While the simple BSP-Tree implementation has problems with intersecting objects, the A-Buffer implementation can handle such scenes without problem, as shown in Figure 6.2.

6.5 Comparison OIT vs BSP trees in the Framework

Both methods were implemented in the framework and OIT seems to be the superior choice in any way. The reasons are explainend in the following subsections.

Implementation difficulty

OIT implementation is way easier than BSP-Trees. To make OIT work just a few methods need to be added to the framebuffer. When `setPixel` is called, a transparent value needs to be put in a list instead of directly writing it into the framebuffer. After all objects are rendered the list needs to be sorted and blended together. List handling is very easy in java since the collection API already supports sorting when the "comparable" interface is implemented. Blending two colors together isn't difficult either. All this methods could be implemented by students for a task without much a problem.

The BSP implementation does not need a lot of changes in the framebuffer, just a simple blend method if a value has transparency instead of overwriting the old color. The main implementation is in a preprocessing stage before the actual rendering is called. The `Scene` class' draw method loops over all objects and calls the `Renderer` with them. When a model has no BSP-Tree a new one needs to be generated. The BSP tree generation can be integrated here most easy, since it's already available to students. Calling BSP-Tree generation could be done after loading/generating a mesh, but then the sources of those classes should be accessible for the students. But it doesn't really matter where it is called, it just needs to be done before the actual rendering process. The BSP-Tree generation code itself is way more complex than A-Buffer blending code. A big problem is that at this stage the framework does not support anything helpful since methods like clipping are all implemented for the `Vertex` class which is generated from a `Mesh.Vertex` after applying the vertex shader. `Mesh.Vertex` represents a vertex inside a mesh and is a different class than `Vertex` which is used in the rendering process. For the `Mesh.Vertex` class not a lot is implemented so a new clipping code needs to be created. Calculating triangle equations is also required to be added. The whole recursive tree generation is more complex than the other method too. It is unlikely that something from the tree generation can be used as task for students in an introductory course.

Functionality, Errors and Limitations

OIT does always work except, when two polygons are nearly exactly on the same depth, but this is a case where there is no deterministic solution on how this could be solved. So A-Buffer blending should always work no matter what model is loaded. The maximum list size is the number of polygons. The performance is worst case quadratic because of list sorting.
Since the framework could load any model, its hard to optimize a BSP-Tree. A very large model could result in a stack overflow because the tree is generated recursively. An iterative

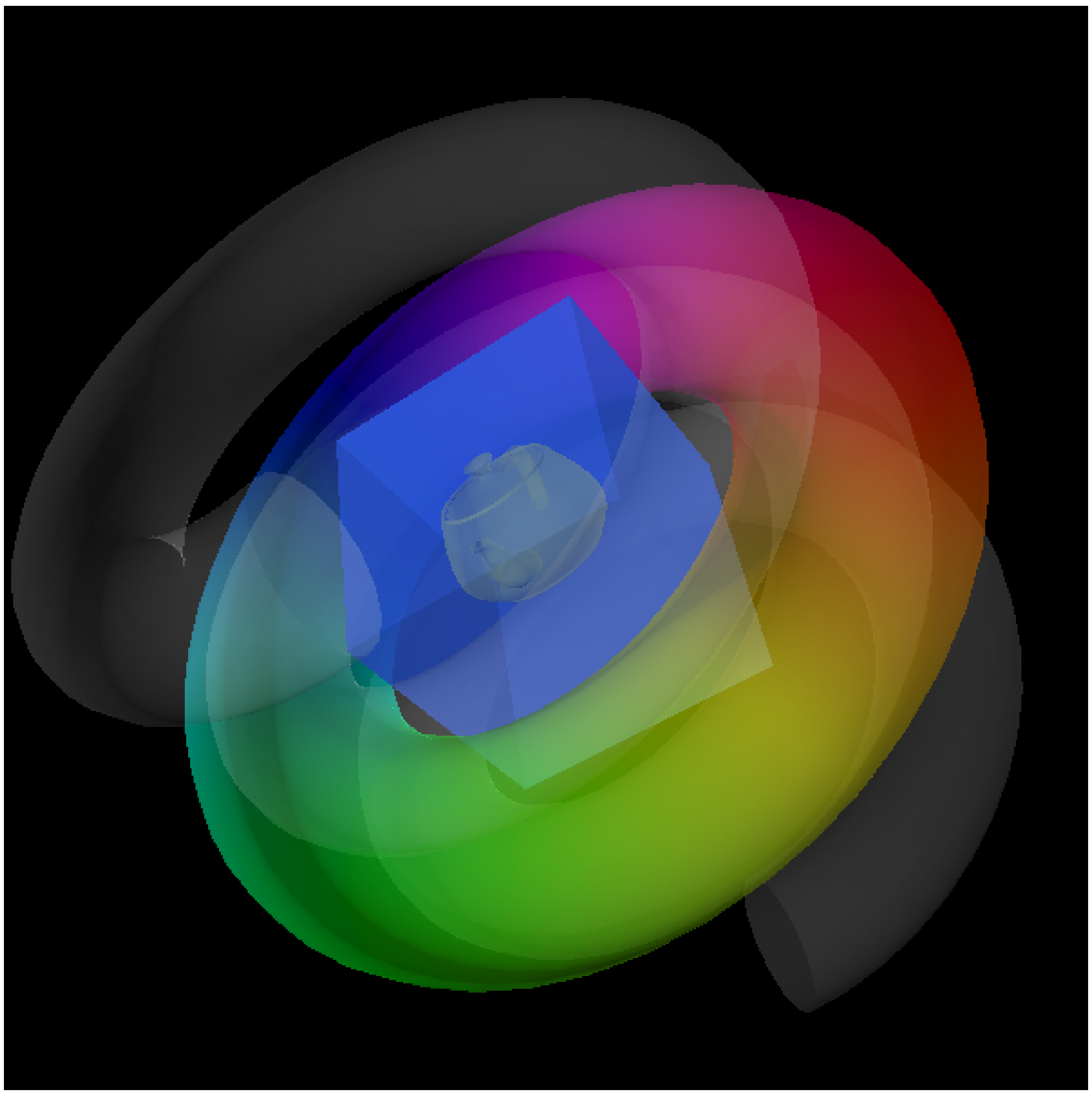


Figure 6.2: Intersecting transparent objects with A-Buffer rendering

implementation would be even more complex. A larger model also causes a noticeable delay for the first frame, on slower computers this can take from multiple seconds up to even minutes. In theory a BSP-Tree should work for all models, but practically it's more error prone than OIT. Optimizing the tree would increase the load time even more, so an arbitrary polygon is used as root node, which can result in very unbalanced trees, but does not really give problems because the whole tree has to be iterated for rendering anyway. It's also way more difficult to test and ensure that the reference code has a desired level of quality and it's very hard to be sure it does not give any problems in some rare special cases. Another disadvantage is that intersecting models can not be solved because combining BSP-Trees would hurt the framerate a lot. This means a way to sort objects needs to be added. The test implementation uses a render order parameter for each object, alternatively sorting according to object's depth works too in most cases.

Performance

OIT does constantly reduce the fps a bit, increased by the amount of transparent planes, while BSP-Trees increase the load the time of a model by a lot. When loading a file or creating a shape, there is no noticeable large delay until the first frame is shown. While BSP-Tree generation with simple models is still fast, it can take a long time when a complex model is loaded. This is also bad because the framework does not support feedback to the user for loading progress. Until now, it was not needed because all shapes and test scenes loaded nearly instantly. On slower computers like a netbook, this generation process can take quite some time like 30 seconds or more.

6.6 Synchronization Problems

Up to now, the frameworks framebuffer was not synchronized which resulted in artifacts sometimes. This is not directly tied to transparency, but with alpha blending implementation the problem becomes a lot more noticeable. The framework uses threading in the rendering process, drawing a polygon is called in an own thread. Up to now, the framebuffer did not have any synchronization which resulted in artifacts when two faces of the same model where overlapping on the image. When backface culling is enabled, which usually is the case because there is not really a reason to not use it during fill rasterization when no transparency is present, there are usually no artifacts visible so it seems this "bug" was overlooked up to now. When two threads want to write a color value at the same pixel it can result in a wrong order. Before the color is written the current depth value is checked. It is possible that two or more threads check the depth at the same time and then write over each other in the wrong order. Since java does support thread synchronization, this problem is easily solved by using synchronization in the `setPixel` method. Of course this comes at a performance cost but this problem should be fixed even without transparency.
Line rasterization suffers even more from this problem. With models like a cube where two

adjacent triangles have very different shading, it is noticeable that some edges are flickering between two different color tones. The problem here is that two triangles share the same edge, their vertices on this edge have the exact same coordinates. This results in a Z-Fighting problem in the framebuffer. When no threading is used, one edge is drawn first and the other with the same depths is discarded, but when those edges are drawn by two different threads on each pixel of this edge another thread may be faster. Unfortunately this can't be solved easily by synchronizing `setPixel`. Every pixel which is covered by the same line would need to be locked when this line is drawn. This would be very complex to handle and very likely cost more performance than disable threading for line rasterizers. So the current solution is to not use threading for wireframe mode. It is still faster than fill rasterization so it is not a big problem. A better solution would very likely be a lot of work.

Benchmarks

Finally it seems interesting to get some actual numbers on how well these changes perform. As test system a PC with Windows 7 64bit and Intel I7 4770K CPU has been used (java version 1.7.0_65-b20). Test settings are the frameworks default window size and camera after starting. As test scenes the creatable models Cube, Torus and Helix will be used. It is hard to create a good performance comparison environment for the framework so this seems a feasible reproduceable test situation. As shader setting the Lambert-Phong shader reference implementation is used.
The tested configurations are the current master branch which is an old version without any changes by me, which also does not have a fix for the synchronization problem. The second test configuration is the new framework version with OIT and synchronization fix, but without actual transparent faces. The third test is the same as the second one but with 50 percent transparency. Finally the performance of my BSP-Tree implementation is measured. Backface culling is always disabled during these tests. The task variation for the new versions are all set to 0 which equals to the old algorithms.

My results of the performance comparison were (numbers in frames per second, fps):

Scene	Old version	New version	OIT	BSP
Cube	1380	170	99	267
Torus	1150	351	143	134
Helix	417	285	194	48

Table 6.1: Benchmark results

Additional notes: Old version does give clearly visible artifacts. BSP-Tree generation does already cause a slightly noticeable delay with the helix model, it is about half a second.

It is clearly visible that the old version has a superior framerate especially for models with few vertices/triangles, but unfortunately it does not work well with threading. With active backface culling, artefacts are not that visible but still there. The synchronization fix actually costs a lot of

performance. The new OIT implementations seems to scale a lot more with number of covered pixels than with vertex count. The cube covers a lot more pixels with default position, scaling and camera than the torus and the helix. Both torus and helix are viewed from above which means the helix has multiple transparent layers to blend.
The BSP-Tree implementation is more similar to the old version where vertex/triangle count is the major factor for performance.
This all makes sense, since the OIT method only needs to sort and blend a list for a pixel when something transparent is drawn on it, while a BSP tree becomes more complex with higher polygon counts.

CHAPTER 7

Conclusion

Switching tasks with an ANT-script by replacing classes is possible, but a bit tricky to set up in the script, since ANT without extensions does not support arrays or lists. To overcome this problem, a workaround with internal dummy targets is used. This solution does work, but setting up new tasks is a bit of a hassle, although by copying and adjusting an existing one it is not too much of a problem.
Weiler-Athertons clipping algorithm does not give any advantage for clipping in the framework. Even as alternative task for teaching purposes it does not seem feasible since the effort of implementing special case handling exceeds the normal case by far. While the algorithm has it's advantages over Sutherland-Hodgman they do not apply in this context.
Backface culling is quite easy and adding a different method is not difficult either. The new angle method's implementation is even easier than the old one that uses the signed triangle area. This method could also be used in world space before clipping, but that would make task switching unnecessary complicated.
New shading models can easily be integrated in the framework. Oren-Nayar and Cook-Torrance work quite well and even more complex models could be added without problems. The current restriction lies in the render loop architecture that only supports a single render pass, which makes some complex effects impossible.
The new order-independent transparency implementation works very well feature wise. The performance hit is clearly there but a major reason for this is also the synchronization fix that has been implemented because the framework had rendering artifacts in some cases. With transparency those artifacts where unbearable. In case of the framework (or maybe a software render in general) order-independent transparency works better than BSP-Trees. For a general software based renderer OIT does give good results and the performance isn't worse. This might be dependent on the language, but when a powerful collection API is present like in java it is easy to implemented list sorting and it might be even faster than BSP-Trees in complex scenes.

Finally, I can say that most alternative algorithms to the current ones are very likely an increase in difficulty for most students. The only exception from the methods I tried is backface culling.

The new transparency feature is also not really hard to implement so it should be possible to use this as task for students without many problems.

Bibliography

[1] P. Beckmann and A. Spizzichino. *The scattering of electromagnetic waves from rough surfaces*. Pergamon Press; [distributed in the Western Hemisphere by Macmillan, New York], 1963.

[2] Loren Carpenter. The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, January 1984.

[3] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[4] Heinrich Fink, Thomas Weber, and Michael Wimmer. Teaching a modern graphics pipeline using a shader-based software renderer. In Giovanni Gallo and Beatriz Sousa Santos, editors, *Eurographics 2012 – Education Papers*, pages 73–80. Eurographics Association, May 2012.

[5] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.*, 14(3):124–133, July 1980.

[6] Game Programming Wiki. http://content.gpwiki.org/d3dbook:(lighting)_cook-torrance. Accessed: 18.8.2014.

[7] Game Programming Wiki. http://content.gpwiki.org/index.php/d3dbook:(lighting)_oren-nayar. Accessed: 18.8.2014.

[8] M. Oren and S.K. Nayar. Generalization of Lambert's Reflectance Model. In *ACM 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 239–246, Jul 1994.

[9] Christophe Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13:233–246, 1994.

[10] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, January 1974.

[11] K. E. Torrance and E. M. Sparrow. Radiometry. chapter Theory for Off-specular Reflection from Roughened Surfaces, pages 32–41. Jones and Bartlett Publishers, Inc., USA, 1992.

[12] John Edward Warnock. *A Hidden Surface Algorithm for Computer Generated Halftone Pictures*. PhD thesis, 1969.

[13] Alan Watt. *3D Computer Graphics*. 3rd edition, 2000.

[14] Kevin James Weiler and Peter Atherton. *Hidden Surface Removal Using Polygon Area Sorting*. Cornell University, 1978.

[15] Xiaolin Wu. An efficient antialiasing technique. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 143–152, 1991.