

# **Practical Collision Detection on the GPU**

## **A Case Study Using CInDeR**

Markus Malmsten and Simon Klasén

Department of Computer Science, Lund Institute of Technology, Lund, Sweden

## Abstract

Collision detection in real-time rendering is used in a variety of fields, such as CAD, simulations, robotics and games. It's also one of the main bottlenecks in the real-time rendering loop. This has contributed to the development of GPU-based collision detection algorithms. By moving the execution from the CPU to the Graphics Processing Unit (GPU) an overall speedup can be gained. The GPU has become an extremely powerful and flexible processor and the recent additions of programmability has made new types of implementations possible.

We have implemented and examined CInDeR, which is an algorithm for collision detection on the GPU. CInDeR is an image-space algorithm and the results from the algorithm is color coded into the frame buffer. This opens for applications that utilizes pure GPU implementations. One of the main problems with image-space algorithms is the task of reading back the results to the CPU which is too time consuming on todays hardware. Self-interference is another problem with CInDeR. It arises due to z-fighting which is caused by precision errors by the hardware. It will randomly appear that parts of the wireframe of an object is behind the polygon to which it belongs, or in front of it, when it in fact should be at the exact same depth.

In this report we solve problems with bandwidth limitation and self-interference with the aid of Cg. These are the main problems to overcome to make CInDeR practically usable in real time. To reduce the amount of data that needs to be read back to the CPU, we have formulated an approximative algorithm. The algorithm is centered around successively reducing the buffer size through a filtering strategy similar to that used for texture minification, or mip-mapping. Being approximative, this algorithm will sometimes miss valid collisions. To solve this we propose an alternate strategy which only requires an extra render-to-texture pass. The problem with self-interferences is solved by filtering out the occurrences with another Cg program. To remove self-interferences before they are registered, and thus prevent real collisions from being overwritten, other techniques can be used. This is also discussed. To test CInDeR in a real-time application, demos utilizing the algorithm were made.

The use of Cg opens for many possibilities. For example, the information encoded into the collision result is not limited to purely object or polygon ids. This is demonstrated in a demo where the normal values of the contact points are encoded directly into the collision results. In this case we remove the need for identifying the contact points and looking up the normal values on the CPU. Instead the normal values in the collision results are used directly when calculating the collision response.

Key words: Collision Detection, CInDeR, GPGPU, Cg, Graphics Hardware, Imagespace Computations



# Index

<b>1. Introduction</b>	<b>1</b>
1.1 Problem	2
1.2 Report Structure	2
<b>2. Background</b>	<b>3</b>
2.1 Previous Work	3
2.2 Graphics Hardware	3
2.2.1 Geometry Pipeline	4
2.2.2 Primitive Assembly and Rasterization	4
2.2.3 Fragment Texturing and Coloring	4
2.2.4 Pixel Pipeline	4
2.3 OpenGL	6
2.4 Cg	6
2.5 Python	8
<b>3. Collision and Interference Detection</b>	<b>9</b>
3.1 Algorithm Design According to Problem Classification	9
3.1.1 Model Representations	9
3.1.2 Different Types of Queries	10
3.1.3 Simulation Environments	10
3.2 Different Approaches	10
3.2.1 Feature-oriented	11
3.2.2 Simplex-oriented	11
3.3 Collision Detection on the GPU	11
3.3.1 CInDeR	12
3.3.2 CULLIDE	12

<b>4. CInDeR</b>	<b>13</b>
4.1 The Algorithm	13
4.2 Implementation	14
4.2.1 Detailed Walk-through of the Implementation	14
4.2.1.1 Initialization	15
4.2.1.2 Pass 1: Drawing the Edges	16
4.2.1.3 Pass 2: Virtual Ray-casting - Counting the Front-facing Polygons	17
4.2.1.4 Pass 3: Virtual Ray-casting - Counting the Back-facing Polygons	18
4.2.1.5 Pass 4: Identifying the Penetrating Objects	19
4.2.1.6 Pass 5-7: Virtual Ray-casting - Identifying the Penetrated Objects	20
4.3 Problems With CInDeR	21
4.3.1 Self-Interference	21
4.3.1.1 Offset Edges	22
4.3.1.2 Filtering	22
4.3.1.3 Prevent Self-interference Registration	23
4.3.2 Optimizations	23
4.3.2.1 Off-Screen Rendering	24
4.3.2.2 Mip-mapping	24
4.3.2.3 Occlusion Query	24
4.3.2.4 Further Optimizations	25
4.4 Application Interfacing	25
<b>5 Demos</b>	<b>27</b>
5.1 Stairs	27
5.2 Snow Bunny	28
5.3 Coloring the Edges	30
5.3.1 Changes to the Algorithm	30
5.3.2 Added Approximation	31
<b>6. Result</b>	<b>33</b>
6.1 Acknowledgments	34
<b>7. References</b>	<b>35</b>
<b>Appendix A</b>	<b>37</b>

# 1. Introduction

The field of computer graphics can be traced back to the 1940's and the projects Whirlwind and SAGE, which were designed to support the American military defense [1]. The Whirlwind computer was originally intended to be a part of a general-purpose flight simulator, but it evolved into the first real-time, general-purpose digital computer. SAGE, or Semi-Automatic Ground Environment, was a project aimed at safeguarding the United States, and through it the Air Force became the major sponsor of Whirlwind. Production versions of the Whirlwind computer were designed in a cooperative effort between MIT and IBM and purchased by the Air Force in the 1950's. Whirlwind had the first computer-driven display which was a cathode-ray-tube (CRT) and it displayed vector graphics.

In the 1960's hardware was still expensive and the displays were vector based. About 1965 IBM released the first widely available interactive computer graphics terminal [2]. It had a vector display and sold for more than \$100,000. The next big landmark was a special type of CRT - DVST (direct-view storage tube). It was developed by Tektronix and together with a keyboard and mouse it was sold for \$15,000 in 1968. This made graphics affordable. The wish to develop more realistic flight simulators in the end of the 1960's required solid colored surfaces, and this motivated the development of raster displays. Systems built by GE for NASA were probably the earliest examples of such displays.

The Pong video arcade game came in 1972 and the first film with computer graphics, "Westworld", came in 1973 [3]. The 1970's also saw contributions such as the Gouraud and Phong shading methods [4], texture mapping, Z-buffer hidden surface algorithm, environment mapping and bump mapping. The computer Apple II came in 1977 [5]. It was the first computer with a color display. It also came in a case with a keyboard and the BASIC programming language built in. This made it the first real user-friendly system.

All major film studios had created a computer graphics department in the early 1980's and some of the first feature-length movies to include digital animation, such as Tron (1982) and The Great Mouse Detective (1986), were made. Because memory was too expensive, it wasn't until the 1980's that high resolution raster displays became feasible. The first graphics standard, GKS, came in 1985 and the 3D extension PHIGS became ANSI standard in 1988. IBM introduced the Video Graphics Array, VGA, in 1987 and SVGA followed in 1989.

With the introduction of VGA and SVGA, personal computers could display photo-realistic images and movies. In 1992 the OpenGL specification was introduced by SGI. The very successful film Jurassic Park, which contained much computer graphics, came in 1993 and the first full-length, computer-generated feature film, Toy Story, came in 1995. In 1996 Microsoft introduced DirectX [6]. The Advanced Graphics Port, AGP, which was designed by Intel for video cards was released in 1997. The Graphics Processing Unit, GPU, was introduced by Nvidia in 1999 as a single chip processor located on the video card.

Since the introduction of the GPU it has evolved and become an extremely powerful and flexible processor. The number of transistors in today's GPUs are in the order of the latest CPUs. Comparing for example the ATI Radeon X850 which has around 160 million transistors [7] with the latest Pentium 4 600 series with its 169 million transistors [8]. With the development of programmable graphics hardware a new tool exists to help developing more elaborate shader programs. Both the vertex and the pixel processing support vector operations with full floating point precision. High level languages have also been developed aiding in the use of the GPU for more general tasks than just the actual rendering, such as robot motion planning or computation of Voronoi diagrams. When taking advantage of the GPU for calculations with other purposes than just graphics operations we talk about General Purpose computations on GPUs, GPGPU.

## 1.1 Problem

Collision detection in real time rendering is used in many areas such as CAD, simulations, robotics and games. Depending on the model representation, query types and the simulation environment, different algorithms are used which are optimized for specific circumstances. Conventional collision detection is performed on the CPU using various methods such as utilizing bounding boxes. It is one of the main bottlenecks in the real-time rendering loop. Therefore improvements in this area is of great interest and much research has been done in the field.

With the latest GPUs, rendering polygons is no longer the main bottleneck in the real time rendering loop for many applications. The excessive capacity of the GPU can be used to aid other tasks such as collision detection. This could result in gained performance, both in terms of faster execution and freed up CPU resources.

In this paper we will look into how collision detection can be performed in practice on the GPU. The original version of CInDeR suffers from problems such as self-interferences and bandwidth limitations. Focus will be put on these two problems because they are the most critical in making the algorithm practically usable. Other less critical problems are the limitation of one collision per pixel and errors that can occur due to clipping. We will utilize Cg which is a high-level language for GPUs as a part of our implementation. The use of Cg opens for many possibilities. For example a wide range of information can be encoded into the collision result, these possibilities will be examined.

## 1.2 Report Structure

Chapter 2 presents previous work done in the field and also contains a walk-through of today's graphics hardware and the supporting programming languages we use. Chapter 3 presents the idea of collision detection and also different types of algorithms. Chapter 4 describes the CInDeR algorithm and our implementation of it. It also presents some of the problems with this algorithms and our attempts to solve them. Chapter 5 shows a few of the demos we made to demonstrate different aspects of the algorithm, and we discuss the benefits and difficulties. In chapter 6 we present our results and conclusions.

## 2. Background

### 2.1 Previous Work

Typically collision detection is performed in two stages. First culling is performed to reduce the number of potentially colliding objects and thus less number of pairwise tests has to be made. Next the pairs of objects in proximity are checked for collision. The problem formulation of collision detection can look very different depending on the model, the environment and what kind of information that is wanted. For an extensive survey we suggest [10, 11].

To accelerate collision detection, different methods using graphics hardware has been researched. CInDeR is an image-space interference detection algorithm that utilizes the stencil and depth buffers, and as all algorithms doing so, it is limited to closed objects. There is a brief summarize of the methods leading up to and inspiring the development of CInDeR in [9].

### 2.2 Graphics Hardware

The main function of the graphics pipeline is to render a two-dimensional image given a virtual camera, three-dimensional objects, light sources, textures and more. The graphics hardware can be viewed as a streaming processor where sequential data is processed in a near identical manner through all the stages of the pipeline. Figure 1 below shows an overview of the pipeline. The first stage is the geometry stage where per-polygon and per-vertex operations take place. This stage's output consists of transformed and projected vertices, colors and texture coordinates. The rasterizer then transforms these polygons into fragments in image space and they pass through different interpolation and test stages in the pixel pipeline. The result is written to the frame buffer. We won't go into details of the workings of the pipeline in this report, but an overview of the stages.

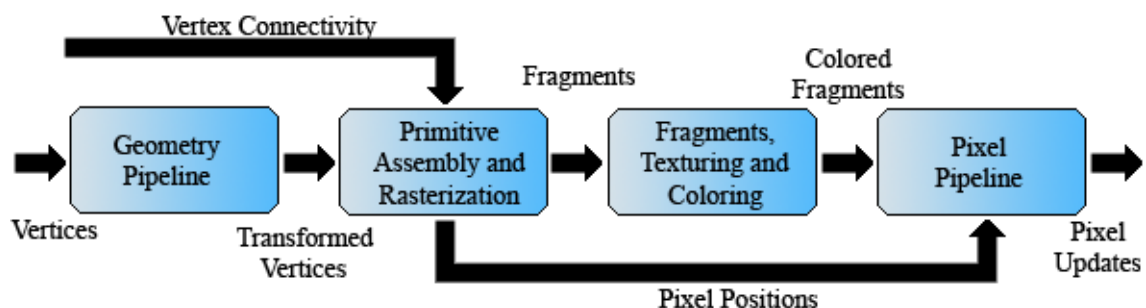


Figure 1: Overview of the graphics pipeline



### 2.2.1 Geometry Pipeline

The geometry pipeline performs per-vertex operations. The stages of the geometry pipeline are shown in Figure 2. First comes the Model & View transformation. The model transform positions and transforms the models from their individual model spaces into the unique world space. The view transform then transforms the world coordinates with regards to the camera and the resulting space is called the eye space. Both the model and the view transform are implemented as 4x4 matrices. For efficiency reasons these are usually concatenated into a single matrix. Lighting, projection and clipping are then performed.



Figure 2: The geometry pipeline

### 2.2.2 Primitive Assembly and Rasterization

In this stage the vertices are connected and rasterized into fragments. The transformed vertices flow in sequence into this stage accompanied by information that determines if they belong to triangles, lines or points. Culling is then performed on these primitives. This includes both clipping to the view frustum and discarding of primitives based on if the face forward or backward. The remaining primitives are then rasterized according to their respective rules and a set of pixel locations and fragments are the result. A fragment is a potential pixel, it is determined in the pixel pipeline whether it's actually drawn or not.

### 2.2.3 Fragment Texturing and Coloring

This stage determines the fragments final color by interpolation, texturing and coloring. In addition this stage may also determine a new depth or even discard the fragment entirely.

### 2.2.4 Pixel Pipeline

This stage performs per-fragment operations before updating the frame buffer. These operations are a standard part of OpenGL and Direct3D. The different stages are shown in Figure 3. If any of the tests fail, the fragment is discarded and the pixel remains unchanged, although a stencil write operation may occur.

The pixel ownership test allows the window system to control the GL's behavior and possibly discard the fragment if for instance the pixel isn't owned by this GL context. An overlapping window will have this affect. The scissor test determines if the pixel lies within the scissor rectangle which is defined by a left, bottom, width and height value. The alpha test only applies in RGBA mode and discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value. The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at the fragments location and a reference value. The depth buffer test discards the incoming fragment if a depth comparison fails. The stencil value at the fragments position is updated according to the function currently in effect for depth buffer test failure. The scissor, alpha, stencil and depth tests can all be enabled and disabled. If disabled the fragment always passes.

Blending combines color values of the incoming fragments with the color values stored in the frame buffer at the fragments position. Dithering selects between two color values or indices. This selection may depend on the coordinates of the pixel and the color of the fragment. Finally a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the frame buffer. The result is written to the frame buffer. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of it's value.

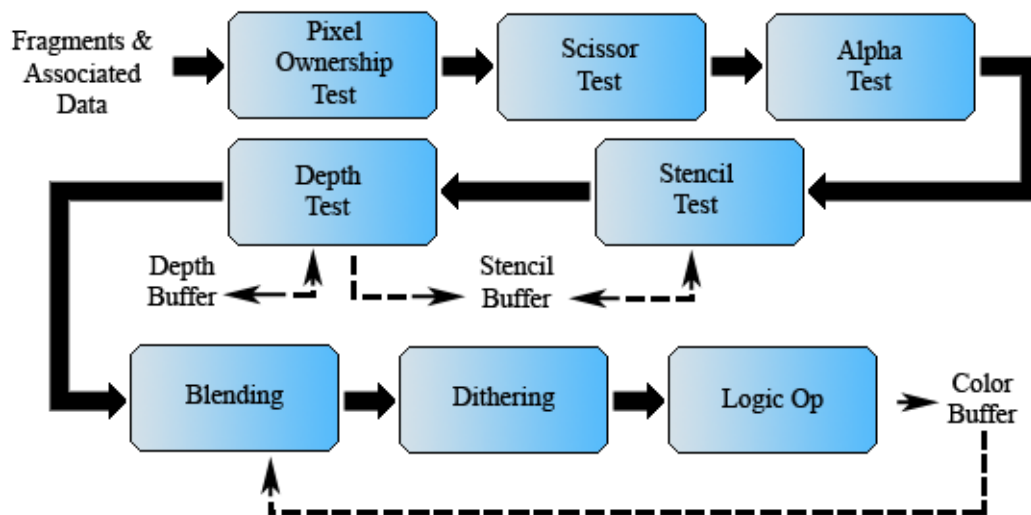


Figure 3: The pixel pipeline

## 2.3 OpenGL

OpenGL is a widely used graphics application programming interface (API), i.e. it's a software interface to graphics hardware [14]. It's not a programming language like C or C++, but a library, which provides some prepackaged functionality [15]. OpenGL is a procedural graphics API. Instead of describing the scene and how it should appear, the programmer prescribes the steps necessary to achieve a certain appearance or effect. All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system. The backward compatibility ensures that existing applications do not become obsolete. The other main graphics API is DirectX, but it is only developed for Windows and Xbox, where version 8.1 of DirectX was used as a basis for the Xbox console API. The functional differences between DirectX and OpenGL are however small.

Although the OpenGL specification defines a particular graphics processing pipeline, platform vendors have the freedom to implement it how they see fit, in both hardware and software. This makes OpenGL available in the full range of computers, from low-cost PCs to supercomputers.

In the programmers perspective, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions. Additional commands control how these objects are rendered into the frame buffer. There are also calls to control of the frame buffer directly, such as reading and writing pixels.

## 2.4 Cg

With the development of programmable graphics processors the need for a high-level programming language arose. Until recently one was limited to fixed-function pipelines, which were configured by setting states with the aid of for example OpenGL. The first programmable pipelines were interfaced at the assembly language level. In practice they were difficult to use efficiently. This motivated the development of Cg, "C for graphics". Cg is based on C, but with enhancements and modifications to make it easy to write programs that compile to highly optimized GPU code. The difference between Cg and C boils down to the difference of the programming models for GPUs and CPUs. GPUs have at least two programmable processors, the vertex and the fragment processor. These two processors and other non-programmable hardware units are linked through data flows according to Figure 4. The program is executed repeatedly on the GPU, once for each element in the data stream. On CPUs on the other hand, the main program is executed only once and CPUs normally have one programmable processor.

The programs in their textual form need to be compiled before the GPU can execute them. In a general-purpose language, the operating system invokes the main routine and the program executes the code contained in that main routine. When the main routine returns, the program terminates. However in Cg the program isn't invoked and run until it terminates, as it would in C or C++. Instead, the Cg compiler translates the program into a form that the 3D API can download to hardware. It's up to the application to call the necessary Cg runtime and 3D API routines to download and configure the program for use by the GPU. After the program is bound it will be run once for each vertex or fragment in the pipeline.

When compiling a program, two things need to be specified in addition to the program; the name of the entry function and the profile name for the entry function. The choice of profile depends on the 3D API used, the hardware capabilities of the GPU and whether it's a vertex or fragment program.

The Cg compiler first translates the Cg program into a form accepted by the application's choice of 3D programming interface, either OpenGL or DirectX. The application then transfers the translation of the Cg program to the GPU using the appropriate 3D API commands. The OpenGL or DirectX driver performs the final translation into the hardware-executable form the GPU requires. Only one program can be loaded at a time in the vertex and fragment processor respectively. However the application may change the current program as needed.

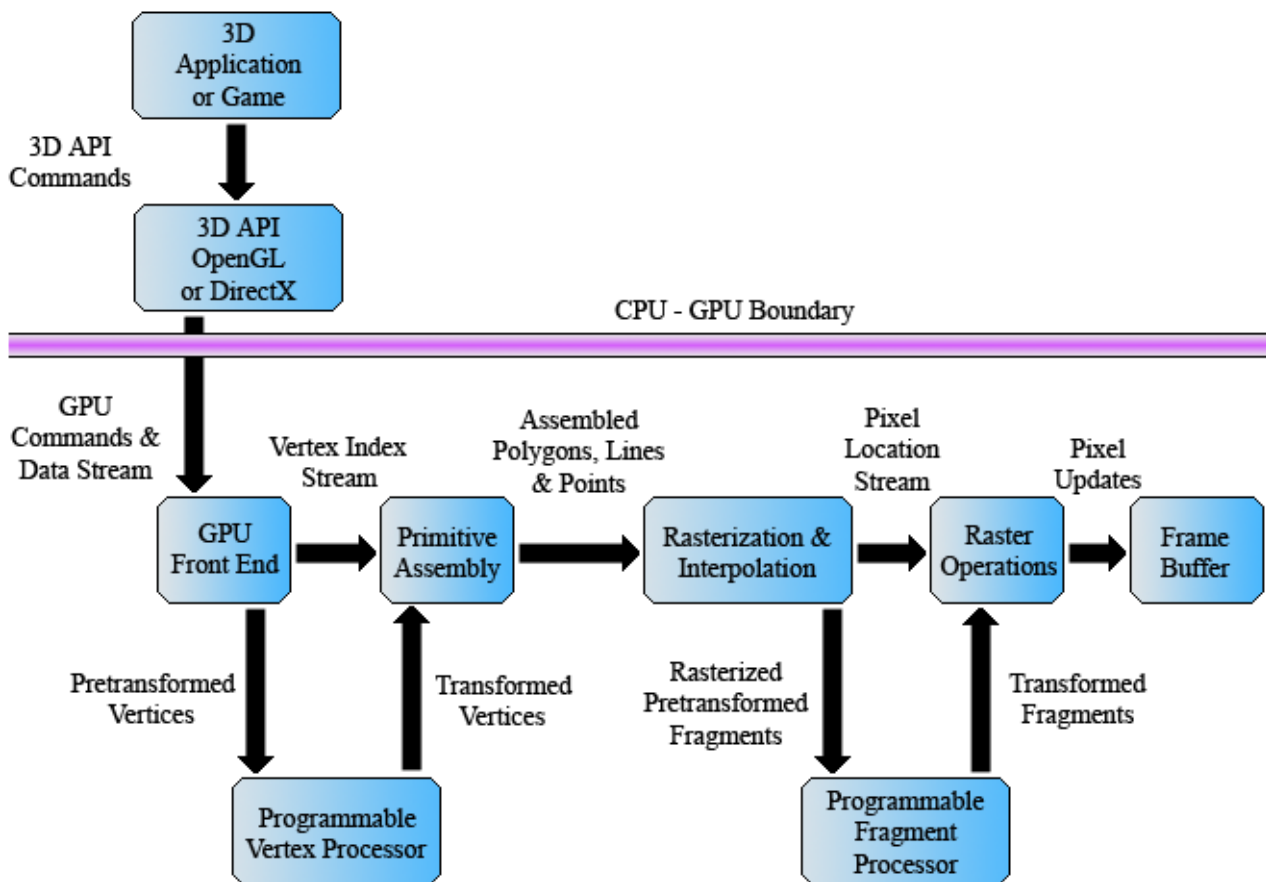


Figure 4: Cg pipeline model

## 2.5 Python

Python [16] is the language we implemented both CInDeR and our demos in. It's an interpreted, interactive, object-oriented programming language. Python combines power with very clear syntax. It has classes, exceptions, very high level dynamic data types, dynamic typing and modules which can be written in C or C++. The language comes with a large standard library that covers areas such as string processing, Internet protocols, software engineering and operating system interfaces.

Version 2.4 was just being released as we started our work but we decided to stick with the 2.3 branch for various reasons. Since version 2.4 was new a few critical packages were not yet supporting it. Also we found 2.3 to work to our satisfaction. The packages required for our implementation are Numeric, PyOpenGL, RenderTexture [27] and PyCg. The computer also needs to have GLUT and the Cg runtime library installed. The Cg runtime library enables programs to be compiled and uploaded during execution of the application. PyCg is the package that binds these functions to Python.

### 3 Collision and Interference Detection

Collision detection is in many applications considered a major bottleneck and is therefore an interesting research area. Many attempts to solve the problem have been made during the years with different approaches. In this chapter we give a brief overview of the problem. For a more extensive survey we recommend Lin and Gottshalk [10] and Jiménez, Thomas and Torras [11].

#### 3.1 Algorithm Design According to Problem Classification

The goal of collision detection is to automatically report a geometric contact when it occurs, is about to occur, or has already occurred. When designing an algorithm for collision detection there are different approaches more or less suitable depending on the problem classification. Three important things that the designer must have in mind is model representation, the desired query types, and the simulation environment.

##### 3.1.1 Model Representations

3D models can be divided into polygonal and nonpolygonal models. Polygonal models are the most commonly used models in computer graphics. They have a simple representation and hardware-acceleration of rendering is widely supported. The polygonal models can be either structured, like a fan or a triangle strip, or a “polygon soup” depending on if they are geometrically connected.

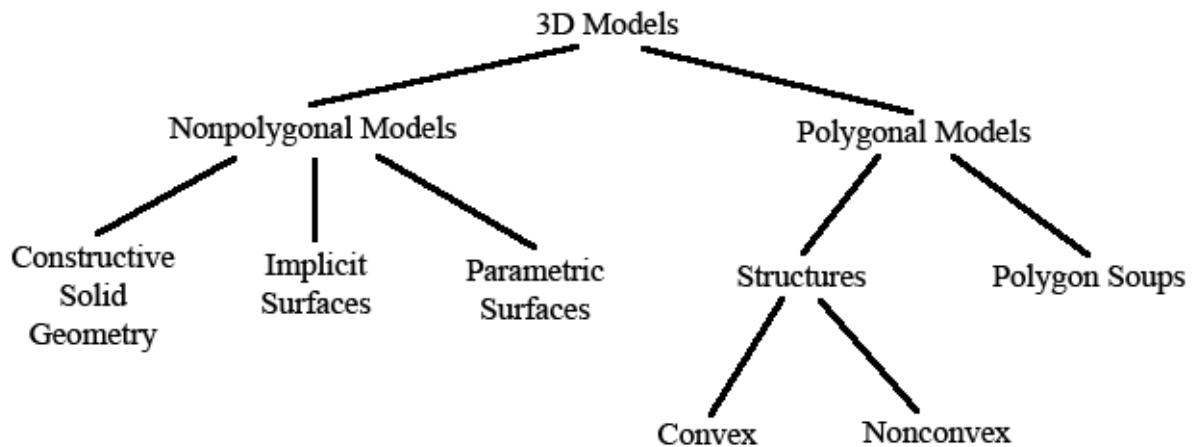


Figure 5: A Taxonomy of 3D Model Representations

The other branch of 3D models are the nonpolygonal. CSG (Constructive Solid Geometry) objects are formed of solid primitives such as blocks and spheres, combined together with a set theoretic operations such as union and intersection. Surfaces are another nonpolygonal representation defined by mappings between space, planes and numbers. Figure 5 is from [10] where a more detailed taxonomy of different 3D models is given. We will exclusively deal with structured polygonal models.

### 3.1.2 Different Types of Queries

Different applications need different types of queries. In the simplest case, we want to know whether two models touch. We may also want to know which parts (if any) touch, i.e. find their intersection. Other times we want to know their separation: if two objects are disjoint, what the minimum Euclidean distance between them is. Further we could be interested in if they penetrate or what the minimum translational distance required to separate them is. We might want to calculate when two objects next collision will be, estimated from their placements and motions, also called ETA computation from "estimated time of arrival".

Distance information is useful for computing interaction forces and penalty functions in robot motion planning and dynamic simulation. Intersection computation is important for physically-based modeling and animation systems which must know all contacts in order to compute collision responses. The ETA solution can be used to control the time step in a simulation.

### 3.1.3 Simulation Environments

The complexity of the problem is much related to how many objects that are being tested for collision. In the simplest case there are just two static objects. When adding more objects, a naive pairwise processing algorithm quickly becomes too time consuming, as it increases quadratically with the number of objects. Many algorithms therefore uses an early rejection test to filter out objects that certainly won't collide. One common approach is to give objects a bounding volume that is examined instead of the object itself. If the models have dynamic movements, rotations and translations, rejection test can be based on velocities and accelerations of objects. Different kind of data types are also common to make rejection tests possible. These are often trees that divide the objects spatially. Rejection tests give great speed-up as long as there aren't any collisions since more exact collision tests can be omitted.

## 3.2 Different Approaches

Depending on the design classification, a decision has to be made towards which type of algorithm that is to be implemented. Most modern collision detection schemes deal with convex, polygonal shapes with a high polygon count, and seek to return very accurate collision details such as penetration depth and contact points [17]. With these kinds of constraints there are two common approaches taken: feature-oriented and simplex-oriented collision detection schemes. Feature-oriented being based on the Lin-Canny algorithm [18] and simplex-oriented being based on the GJK algorithm [19]. For an overview of other algorithms see [20].

When relaxing the constraints on a collision detection system, other methods can be used. Most RTS (Real Time Strategy) games for example, only use radius based methods or AABB (Axis Aligned Bounding Boxes) for collision detection, and thus trade accuracy for speed.

### 3.2.1 Feature-oriented

The method works by finding and maintaining the pair of closest features (vertex, edge, or face) on two polyhedra. Lin-Canny is the classic algorithm which introduced this method [18]. By tracking closest features, different culling strategies can be applied. In practice this results in a near constant running time. The algorithm works by first pre-processing convex polyhedra to find their Voronoi regions. For an object, a Voronoi region of a feature is a set of points which are closer to that feature, than to any other feature on that object. The main part of the algorithm searches for closest points between a pair of features on two objects. The confirmation is done by checking that each point lies within the Voronoi region of the other feature. When the closest point distance falls below a small threshold it's assumed that a collision has occurred. Lin-Canny has spawned many variations and improvements, including some which overcome the algorithm's Achilles heel: its inability to elegantly handle interpenetrating shapes.

### 3.2.2 Simplex-oriented

A  $d$ -simplex is the convex hull of  $d+1$  affinely independent points in  $d$ -dimensional space. A Simplex is a  $d$ -simplex, for some given  $d$ .

The Gilbert-Johnson-Keerthi (GJK) algorithm is a simplex-based algorithm that, given two sets of vertices as inputs, finds the Euclidean distance (and closest points) between the convex hulls of these sets. The algorithm is based on the fact that the separation distance between two polyhedra  $A$  and  $B$  is equivalent to the shortest distance between their Minkowski difference  $C$ , and the origin. Combinations of points belonging to  $C$  is then used to create convex sets that have a convex hull forming a simplex inside  $C$ .

Initially an arbitrary point  $P$  in  $C$  is chosen as a starting simplex. If  $P$  is the origin itself, then  $A$  and  $B$  intersects. Otherwise the algorithm will iteratively form new simplices, that is ensured to contain a point closer to the origin than found in earlier steps. If there are no intersection between  $A$  and  $B$  the algorithm will terminate with the shortest distance between  $A$  and  $B$  as result, in a finite number of steps.

For an explanation of the Minkowski difference and a more detailed paper of GJK see [19].

## 3.3 Collision Detection on the GPU

There has been many algorithms presented that utilizes the GPU for collision detection [9, 21, 22, 23, 24, 25, 26]. At a broad level these can be classified into two categories: techniques for using the depth and stencil buffers to compute interference [9, 21, 22, 23, 24], and fast computation of distance fields for proximity queries [25, 26]. These algorithms suffer from different limitations such as bandwidth limitations, restriction to closed objects and only allowing pair-wise checking. The algorithms that perform image-space calculations has the additional limitation of the resolution of the viewport and the depth buffer which affects how precise the detections can be made.



### 3.3.1 CInDeR

Traditional ray-casting is an algorithm that's used to render scenes by following rays of light backwards from the eye of the observer to a light source. CInDeR uses frame buffer operations to implement a virtual ray-casting algorithm which detects static interference between solid polyhedral objects [9]. It writes the edges of the objects to the depth buffer and by virtual ray-casting it checks which objects they penetrate. Virtual ray-casting in this context is a technique that locates a point relative to a solid by casting a semi-infinite ray from the point. The number of polygons that the ray passes through are counted, see figure 6. An even sum means that the point is outside the solid and an un-even sum means that it is inside. CInDeR uses the stencil buffer for counting the number of front and back facing polygons the rays pass through on the way from the edge to the near clipping plane, and stores the colliding objects id's in the color buffer. The algorithm doesn't require any preprocessing or special data structures and the tests are performed on the geometry itself, in image space. The algorithm's expected running time is linear in both the number of objects being tested and the number of polygons comprising the objects.

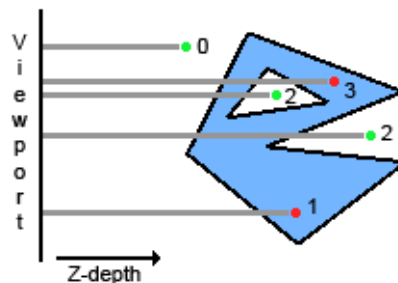


Figure 6: Determining a points location relative to a solid by virtual ray-casting

### 3.3.2 CULLIDE

The CULLIDE algorithm supports collision detection between multiple deformable and breakable objects [21]. It computes a potentially colliding set (PCS) and use visibility computations to prune this set. The visibility relationships are computed using image-space occlusion queries. Given a set S of objects, the relative visibility of an object O with respect to S is tested. The query checks whether any part of O is occluded by S. It rasterizes all the objects belonging to S. O is considered fully-visible if all the fragments generated by rasterization of O have a depth value less than the corresponding pixels in the frame buffer. The accuracy of the algorithm is governed by the underlying precision of the visibility query. The pruning of the PCS takes place in two stages, one at the object level and one at a sub-object level. The exact triangle-triangle intersection tests are then performed on the CPU.

## 4. CInDeR

The CInDeR algorithm which we chose to implement uses the technique of virtual ray-casting. It uses frame buffer operations to detect static interference between solid polyhedral objects. The algorithm's expected asymptotic running time is linear in both the number of objects and number of polygons in the models. It requires no preprocessing or special data structures and it can handle both convex and non-convex geometry with hollow regions. The interference tests are performed on the geometry itself and not on an approximation of the surface.

Two objects are detected as interfering if and only if an edge of one object intersects the volume of another object. The way this is detected is by virtual ray-casting and counting the number of polygon faces that are being passed through. This is done for every pixel.

Since the algorithm is image based it can only detect collisions, e.g. interference, in the frame being analyzed. We must accept that collisions that are about to happen or that have already occurred won't be detected. This could occur when a thin part of one object passes through a thin part of another in such a way that they pass through each other in the space of one frame. The thickness required to eliminate such events grows with the speeds of the objects involved.

### 4.1 The Algorithm

The algorithm, see Appendix A, finds in which pixels there are intersections and also which objects that are intersecting. It is divided up into different passes. Each pass draws all the objects in the scene and by applying different settings to the graphics pipeline, the virtual ray-casting is performed, see section 3.3.1. In the first pass the objects are drawn as wireframes to the depth buffer. It's from these edges that the virtual rays will be cast. All front-facing polygons are then drawn filled and the value in the stencil buffer is increased for each pixel that passes the depth test, i.e. lie in front of the edge. Next the values in the stencil buffer is decreased in the same way for the back-facing polygons. The result is a non-zero value in the stencil buffer where there has been more front-facing polygons than back-facing ones, meaning the edge in that pixel is inside an object.

Now for the identification. The penetrating objects are identified by drawing the wireframes color coded according to the object ids to the color buffer where the stencil value is positive. To identify the penetrated objects, virtual ray-casting is performed, but this time object by object. The stencil buffer is reset and the front- and back-facing polygons are drawn for an object. The stencil buffer now holds positive values in only those pixels where this one object is the penetrated object. The object-id is color coded and stored with the penetrated object in the same pixel. Finally the stencil buffer is reset and the next object is checked for penetrations.

After the algorithms has run, the frame buffer contains the color value (0,0,0,0) in all pixels where there are no collisions. In those pixels where there are collisions we have coded identifiers of the involved objects. The entire color buffer can now be read back to the CPU where suitable collision responses can be computed.

## 4.2. Implementation

We have encapsulated the algorithm into a class which requires a projection function and a list of the objects during execution. It returns a list of ids representing unique colliding pairs which can then be used as one sees fit. The pseudo code for the algorithm is listed in Appendix A.

The algorithm is implemented using Python and OpenGL alone. But instead of rendering to the frame buffer we render to a texture using RenderTexture [27]. This enables easy access to the results for further processing on the GPU.

At the execution of the CInDeR algorithm, a projection function is supplied from the application. This is needed to set up a transform for the RenderTexture as in the regular render context. With `gluPerspective` it is easy to specify the field of view and clipping planes. `gluLookAt` can then be used to define a view transform (camera settings). The benefit of the perspective function is that the scene displayed to the user can be rendered from a different location and angle than the one CInDeR is set up with. Although, in many situations it is preferred to use the same function for both. One example when it is good to use different perspective functions is in the demo presented in 4.1.3. In that demo we are only interested in spheres colliding with a bunny and don't care about sphere-sphere collisions. Then we can set up CInDeR's perspective function to be close to the bunny for good resolution. In the same time we can display the scene from other angles and distances that are interesting to the user, but bad for CInDeR. For instance we could zoom out heavily and see the scene from great distance but keep the accuracy of the collision detection. We don't even have to look at the colliding objects at all.

### 4.2.1 Detailed Walk-through of the Implementation

In this walk-through we will explain and discuss the code. Our implementation have some slight differences to the one presented in Appendix A, but the principles are the same. To make it easier to follow we have images illustrating the content of the stencil, depth and frame buffers at different stages.

The test scene setup consists of three solid boxes of equal size. They have different colors and are on different depths with the red being closest to the viewer. The scene is displayed in Figure 7.

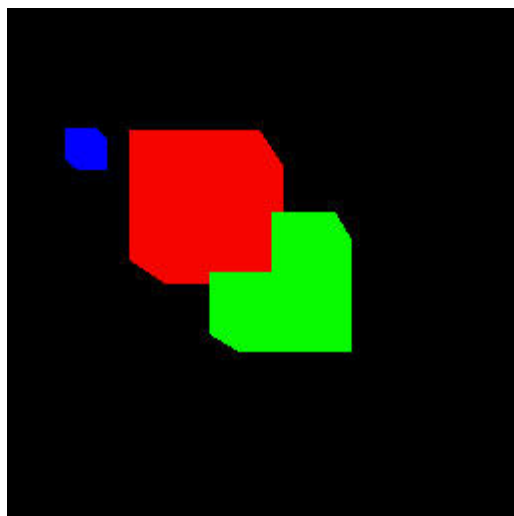


Figure 7: The test scene.

### 4.2.1.1 Initialization

Before the algorithm is executed, a range of RenderTexture objects are created and necessary Cg programs are loaded.

```
1     def cinder(self, objectList, projectionFunc):
```

For each frame the function cinder is invoked with an object list containing all objects in the scene and a projection function.

```
2     self.rendertexture1.BeginCapture()
3     projectionFunc()
```

The RenderTexture is set as the target of rendering and the projection function is then applied to this context.

```
4     glDepthMask(GL_TRUE)
5     glClearDepth(0.0)
6     glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE)
7     glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT | GL_COLOR_BUFFER_BIT)
```

By default writing to the depth buffer and all four color channels is enabled in OpenGL, and the stencil test is disabled. But since we use the same RenderTexture every time, and also following OpenGL practice, we don't assume any states.

Line 4 enables the depth buffer for writing and line 5 sets the clear value for the depth buffer to 0.0, which corresponds to the plane at the same depth as the near plane. The depth buffer is usually cleared to 1.0 when using the z-buffer algorithm for depth tests, but reason for the difference will become apparent later on. Line 6 enables all four color channels for writing in the color buffer and line 7 clears the depth, stencil and color buffers.

```
8     glDepthFunc(GL_ALWAYS)
```

glDepthFunc specifies the function that is used for deciding whether or not a fragment passes the depth test. GL\_LESS is the default value and this was used when rendering Figure 7. This means that a fragment will pass the test if it's closer to the viewer than the previously drawn fragments at the same pixel coordinates. The reason for using GL\_ALWAYS is twofold. We can't use GL\_LESS since nothing would pass the depth test now that we have cleared the depth buffer to 0.0. It also ensures that the resulting value is the last written at a specified coordinate. This will ensure that it's the same object that is being identified later on when we find the penetrating object. This requires that we render the objects in the same order throughout the algorithm. Figure 8a shows what happens when you render the red box before the green with GL\_ALWAYS enabled and Figure 8b shows the green being rendered before the red.

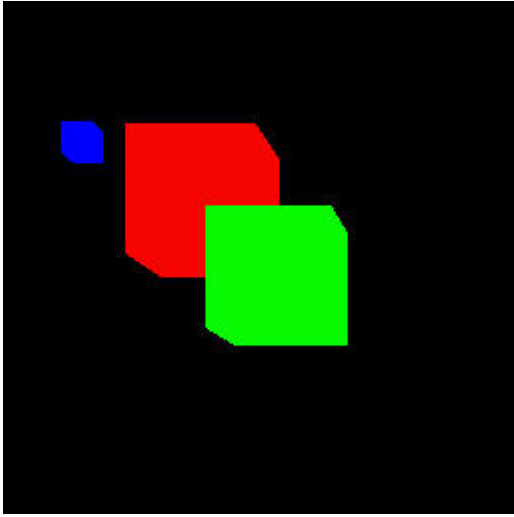


Figure 8a: GL\_ALWAYS, red before green

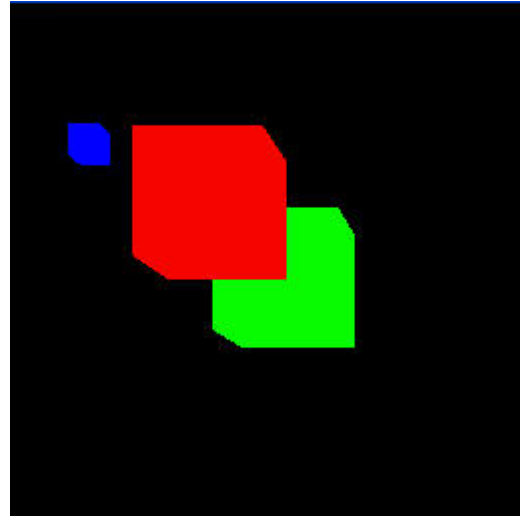


Figure 8b: GL\_ALWAYS, green before red

#### 4.2.1.2 Pass 1: Drawing the Edges

```

9     glEnable(GL_DEPTH_TEST)
10    glDisable(GL_STENCIL_TEST)
11    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)
12    glDisable(GL_CULL_FACE)
13    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)

```

Lines 9-13 applies the necessary settings for the first pass. The reason for enabling the depth test even though we want everything to pass is that we want the depth values written to the z-buffer. However we don't want anything drawn to the color buffer. We then make sure that culling is disabled so every polygon is rendered and finally we set the polygonmode GL\_LINE for all polygons, to render the edges only.

```

# Pass 1
14    for i in range(len(objectlist)):
15        objectlist[i].draw()

```

Now we are ready to make the first pass. A pass is when we draw all the objects in the scene. All objects must have a draw method, which is called as we traverse the objectlist. The parameters in the draw function are used to specify the color coding. At this stage it doesn't matter what color is used since the write access to the color buffer is disabled. The boundary edges of both front-facing and back-facing polygons are drawn as line segments as shown in Figure 9a and they impact the depth buffer as shown in Figure 9b with the values mapped to a gray scale being black at 0.0 and white at 1.0. Only the depth buffer is altered during pass 1. The depth buffer will then remain the same throughout the entire algorithm.

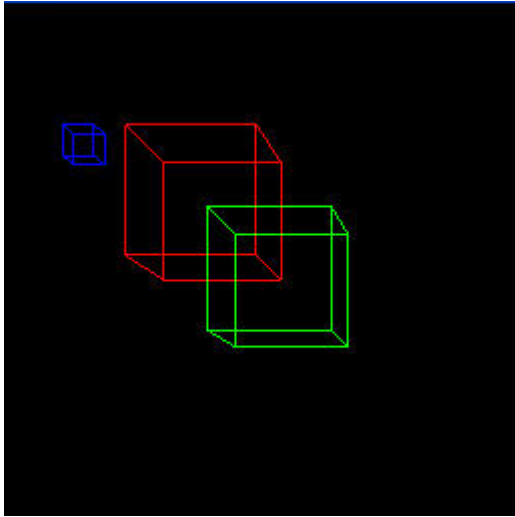


Figure 9a: Boxes drawn as wireframes

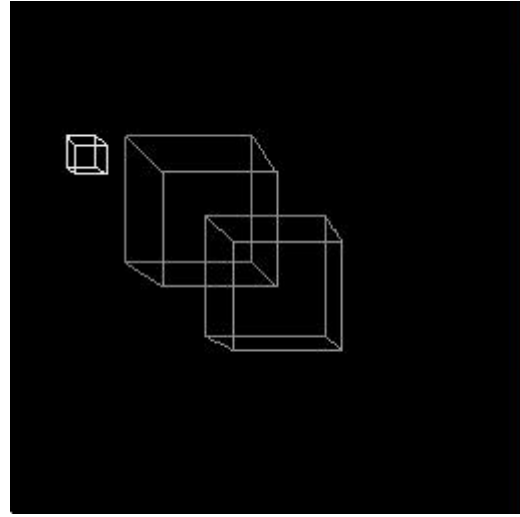


Figure 9b: Depth buffer after pass 1

#### 4.2.1.3 Pass2: Virtual Ray-casting - Counting the Front-facing Polygons

```

16  glDepthMask(GL_FALSE)
17  glDepthFunc(GL_LESS)
18  glEnable(GL_STENCIL_TEST)
19  glStencilFunc(GL_ALWAYS,0,0)
20  glStencilOp(GL_KEEP, GL_KEEP, GL_INCR)
21  glEnable(GL_CULL_FACE)
22  glCullFace(GL_BACK)
23  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)

```

The task now is to perform the virtual ray-casting. Pass 2 involves counting the number of front-facing polygons between the near plane and the edges. Lines 16-23 apply the necessary settings for this. The depth mask is disabled since we want to retain the depth values of the edges. The depth function is set to `GL_LESS` to only count those polygons in front of the edges. We then enable the stencil test and set the stencil function and operation to always increase the value if the fragment passes, which will happen if and only if there is an edge at the same coordinates and the fragment is closer to the near plane. Here it's important that we cleared the depth buffer to 0.0. This will ensure that the stencil buffer will only be increased where there is an edge, and thus a potential collision. Lastly at lines 21-23 we make sure that only the front-facing polygons are drawn, and that they are filled.

```

# Pass 2
24  for i in range(len(objectlist)):
25      objectlist[i].draw()

```

In the same manner as in pass 1 we now traverse the objectlist and invoke the draw method for all objects. The resulting stencil buffer is shown in Figure 10.

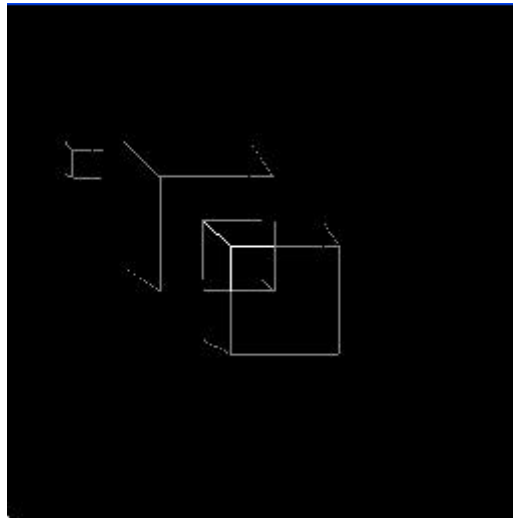


Figure 10: Stencil buffer after pass 2.

#### 4.2.1.4 Pass3: Virtual Ray-casting - Counting the Back-facing Polygons

```

26  glStencilOp(GL_KEEP, GL_KEEP, GL_DECR)
27  glDepthFunc(GL_LEQUAL)
28  glCullFace(GL_FRONT)

```

To complete the virtual ray-casting we need to decrease the stencil buffer in all places where the ray enters an object. This is done in a similar fashion as before but now using the back-facing polygons. First a few settings at lines 26-28. We set the stencil operation to decrease and change the depth function to `GL_LEQUAL`. The reason for the change in depth function is to remove some of the self collisions. Finally we make sure only the back facing polygons are drawn.

```

# Pass 3
29  for i in range(len(objectlist)):
30      objectlist[i].draw()

```

Once again we draw all the objects and the stencil buffer at this point is shown in Figure 11.

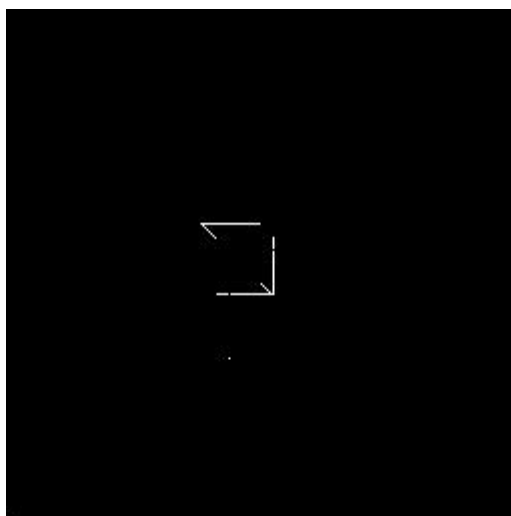


Figure 11: Stencil buffer after pass 3.

#### 4.2.1.5 Pass4: Identifying the Penetrating Objects

```
31  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
32  glDisable(GL_CULL_FACE)
33  glColorMask(GL_TRUE, GL_TRUE, GL_FALSE, GL_FALSE)
34  glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP)
35  glStencilFunc(GL_NOTEQUAL, 0, 127)
36  glDisable(GL_DEPTH_TEST)
```

At all the coordinates where the stencil buffer is non-zero (non-black in Figure 11) we have now found a collision. The next task is to identify which the colliding objects at each position are. The task of identifying the penetrating objects is straight forward. The settings for this pass are shown in lines 31-36. First we make sure the objects are drawn as wireframes, both front and back face. We then enable writing to the color buffer at the R and G channels. This will hold the object id of the penetrating object. We then make the stencil buffer write protected and set the stencil function to allow drawing at any coordinate where the stored value is non-zero. Lastly we disable the depth test.

```
# Pass 4
37  for i in range(len(objectlist)):
38      Red = (0xff00 & (i+1)) >> 8
39      Green = 0x00ff & (i+1)
40      objectlist[i].draw(Red, Green, 0, 0, 1)
```

This pass is slightly different from the previous. For each object we need to encode it's object id for writing into two channels. This is done in lines 38-39. On line 40 we then draw the object using these encoded colors. The stencil and depth buffers remain unchanged but the color buffer now holds the encoded object id's for the penetrating objects. This is shown in Figure 12 below.

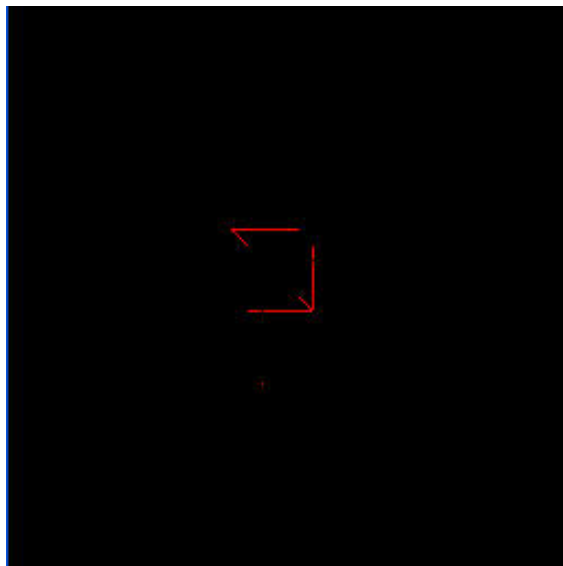


Figure 12: Color buffer after pass 4.



#### 4.2.1.6 Pass5-7: Virtual Ray-casting - Identifying the Penetrated Objects

```
41  glClear(GL_STENCIL_BUFFER_BIT)
42  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)
43  glEnable(GL_STENCIL_TEST)
44  glEnable(GL_DEPTH_TEST)
```

The next step is to identify the penetrated objects. The method is to perform virtual ray-casting like in pass 2-3 but this time object by object. First a few settings. We clear the stencil buffer, set the polygon mode to draw full back-facing and front-facing polygons. Finally we enable both the depth and stencil tests.

```
# Pass 5-7
45  for i in range(len(objectlist)):
46      glStencilFunc(GL_ALWAYS, 0, 0)
47      glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)
48      glEnable(GL_CULL_FACE)
49      glCullFace(GL_BACK)
50      glDepthFunc(GL_LESS)
51      glStencilOp(GL_KEEP, GL_KEEP, GL_INCR)
52      objectlist[i].draw(0,0,0,0,0) # Pass 5
53      glCullFace(GL_FRONT)
54      glDepthFunc(GL_LEQUAL)
55      glStencilOp(GL_KEEP, GL_KEEP, GL_DECR)
56      objectlist[i].draw(0,0,0,0,0) # Pass 6
57      glStencilFunc(GL_NOTEQUAL, 0, 127)
58      Blue = (0xff00 & (i+1)) >> 8
59      Alpha = 0x00ff & (i+1)
60      glColorMask(GL_FALSE, GL_FALSE, GL_TRUE, GL_TRUE)
61      objectlist[i].draw(0, 0, Blue, Alpha, 1) # Pass 7
```

Now for the virtual ray-casting. Lines 46-51 applies the settings needed for the counting of the front-facing polygons just like before in pass 2. Line 52 then draws all the objects, front face only, and increases the stencil buffer where it passes the depth test just like before. Lines 53-55 switches the culling so the back-facing polygons are drawn. It also changes the depth function to match the one used for pass 3 and it changed the stencil operation to decrease. On line 56 we draw the objects once more for the 6th pass. The stencil buffer now contains one in the places where the current object is penetrated by the object already identified in the RG channels. Left is to encode the object id in the same fashion and draw it to the BA channels (lines 57-61). These lines also serve as a clearing of the stencil buffer for the next iteration.

```
62 self.rendertexture1.EndCapture()
```

Finally the capture is ended for the RenderTexture.

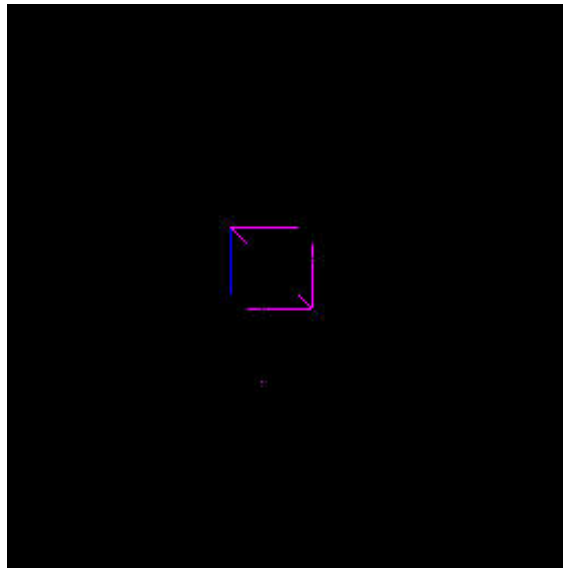


Figure 13: The resulting color buffer containing the object id's

We now have the object id's of the colliding objects for this frame stored in a RenderTexture as shown in Figure 13. The result can now be read back to the CPU where suitable collision responses can be computed.

### 4.3 Problems With CInDeR

CInDeR registers self-interference due to z-fighting. These self-interferences could possibly overwrite real registered collisions because we can only store one collision pair per pixel. Duplicate collision pairs will also be registered when the colliding part of an edge occupies more than one pixel. Both self-interferences and duplicate collision pairs are unwanted and should be removed before returning the results. In chapter 4.3.1 we will discuss these problem and present some possible solutions.

Due to the fact that the memory architecture is optimized for maximum throughput from the CPU to the GPU and not the other way around, it is a very slow operation to read back the complete color buffer to the CPU. To reduce the amount of data that is read back to the CPU and make our implementation practically usable, we implemented a mip-map algorithm in Cg. This algorithm and other approaches to minimize the data is discussed in 4.3.2.

#### 4.3.1 Self-Interference

Self-interference arise due to z-fighting which is caused by precision errors by the hardware. It will randomly appear that parts of the wireframe of an object is behind the polygon to which it belongs, or in front of it, when it in fact should be at the exact same depth. The original implementation incorrectly detects self-interference, the more complex model; higher polygon-count, the more self-interference are reported, see Figure 14 below for an example

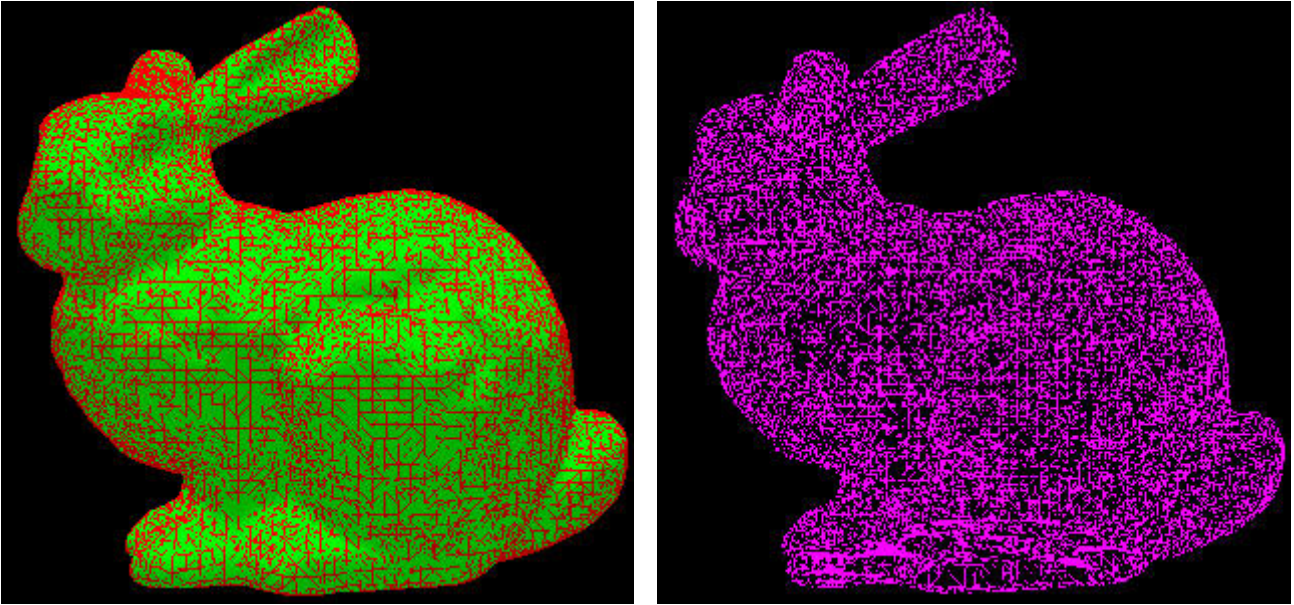


Figure 14: Edges and filled polygons to the left, the resulting self collisions to the right.

There are three main approaches to solve the problem. The self-interference can be filtered out or an offset can be applied, as suggested in [9], when drawing the wireframe to try to minimize self-interference. Even if offset is used there may still remain some self-interference that will need to be filtered out. Self-interferences may block real collisions due to them occupying the same space in the frame buffer. A solution to the problem is to read the pixel that we are currently rendering from the color buffer, and check whether we have a self-interference, and if so discard it. This is however not possible on today's hardware, but can be mimicked by using textures.

#### 4.3.1.1 Offset Edges

The edges could be given an offset manually by making a translation in the direction of the surface normal before rendering the wireframe. It is also possible to use `glPolygonOffset`. Each fragment's depth value will be offset after it is interpolated from the depth values of the appropriate vertices. The disadvantages is that it is hard to find an optimal offset that removes all self-interference. If the offset is too small too much self-interference will remain. The larger the offset, the more missed “true” collisions and the more falsely detected collisions. The optimal offset depends on both the chosen view and the model being rendered.

#### 4.3.1.2 Filtering

Our solution was to allow self-interference, but filter them out before reading back the results. When `CInDeR` returns, the texture with interference information is passed to a Cg shader program for filtering. The interference color reveals both the penetrating and penetrated object in the color channels. If the object identifier in the RG channels are the same as in the BA channels, then self-interference has occurred and the collision is blanked out, i.e. the color is replaced with black `[0,0,0,0]`. This method ensures that self-interference is completely removed.

### 4.3.1.3 Prevent Self-interference Registration

Filtering doesn't completely solve the problem of self-interferences. True collisions can still be overwritten. The best solution to the self-interference problem is therefore to avoid them completely before they are registered, by comparing the color to be written to a pixel with the currently stored color in that pixel. If the two colors represent the same object, it indicates a self-interference and should not be registered. The technique requires the ability to read from the same pixel that we render to. This is not supported by current hardware but there is no apparent reason why this would be a problem to support, as it would not interfere with the parallel architecture of the GPU.

The same effect can be obtained on current hardware by using textures. The technique is to render to a new texture after the identification of the penetrating objects. The first texture holds the object ids for the penetrating objects in the RG channels and the actual test would be performed in a Cg program. When registering the penetrated objects, the fragments BA values will be compared to the RG channels of the first texture. If they are identical the fragment should be discarded and the self-interference will never be written, and thus no real collisions would be overwritten either. If the values are not equal it's a real collision and all four color channels, in the second texture, will be written to; RG taken from the first texture and BA taken from the processed fragment. This technique would require another pass to write the depth values to the second context.

### 4.3.2 Optimizations

When using graphics hardware for non-graphics computation some problems arise. One of the primary ones is retrieving the results. Since all computations eventually are written to the frame buffer, it needs to be read back to the CPU if further processing is to be made. The memory architecture is optimized for maximum throughput from the CPU to the GPU and not the other way around, therefore reading back results is very inefficient. A solution to the problem would be to use the Unified Memory Architecture (UMA), where the CPU and GPU have a shared memory. UMA is already in use in systems such as the Xbox and the architecture is also available for some PC motherboards with built in graphics processors.

PCI-Express is a new technology which allow the CPU and GPU to transfer data faster than earlier existing bus systems, in *both* ways. It's intended as a replacement for both the PCI and the AGP buses. With PCI-Express, which recently has reached the PC market, pixel read back will not be as big of a problem as before.

Both UMA and PCI-Express would improve applications using our implementation of the CInDeR algorithm. With the AGP architecture on our setup, it was needed to speed up the read back of the results to get any usability at all. To decrease the time needed to read back the results we reduced the amount of data by using a mip-map function. Occlusion queries can be used as an initial test to check if there was any collisions at all.

### 4.3.2.1 Off-Screen Rendering

Today's graphics hardware allows a rendering context to be associated with an off-screen rendering surface. An off-screen rendering surface is a part of the frame buffer that is not associated with any visual display. They are commonly used for rendering directly to a texture-map, and that is also the technique we use in our implementation. Reading back pixels to the CPU from the back-buffer imposes a stall in the rendering until the pixels are read. Reading pixels from a texture is also a slow operation, and there is no real speed-up in reading back data to the main memory from a texture compared to the back-buffer. The advantages of rendering to a texture is that the results can be used directly for further computations on the graphics hardware, and also the precision can be set up to 32 bit floats per color channel, compared to 8 bit which is the case for the color buffer.

### 4.3.2.2 Mip-mapping

Mip-mapping is a way of making copies of textures in reduced sizes with the intention of increasing rendering speed and reducing artifacts. To reduce the amount of data that needs to be read back to the main memory we implemented a mip-map function using Cg. This function reads the results from one texture and writes it to a four times smaller one, for as many stages as are specified. For each pixel in the target texture the algorithm chooses the highest value of four pixels in the source texture, thus pixels with no collisions are the least prioritized. This adds another source of approximation since we cannot mip-map down to a pre-determined size and guarantee that no collisions have been lost. Thus this works best if there are few collisions in the same frame and if the concurrent collisions aren't in close proximity. The closer the collisions are together on the screen and the more mip-map stages used, the more likely we are to lose collisions in the process. There is the possibility to use occlusion queries to avoid losing information in the mip-map process.

### 4.3.2.3 Occlusion Query

Occlusion queries is a way to determine the number of pixel writes. In the original paper about CInDeR [9] occlusion queries were used to avoid sending data back when there are no collisions, or when there is only one collision pair to identify. The problem of identifying multiple collision pairs however still remain. Our motivation for using occlusion queries is to reduce information loss during mip-mapping. With occlusion queries we can choose different mip-map strategies depending on the result. If there were only a few pixels in the original texture that passed the occlusion test, then we could mip-map in more steps to produce a much smaller texture, without losing much or any information. The opposite would apply if there are many collisions. Another more sophisticated solution would be to mip-map the texture as long as the same number of unique collision pairs remain. This can be accomplished by using a color to identify information loss and checking for that color after every mip-map stage using occlusion queries. If the color is present the occlusion query would return a positive number and we interrupt the algorithm and return the previous mip-map level.

#### 4.3.2.4 Further Optimizations

More sophisticated methods of processing the results on the GPU can be imagined. By reordering the results in a texture rather than using a mip-map function, all registered collisions can be contained in a smaller area of the texture. As a result less data needs to be read back without losing information.

Encoding other information than pure object or polygon ids is another way of saving computations. Instead of looking up the information you want from the object or polygon by finding the collision point or similar, some information can be encoded directly in the color buffer and obtained as a collision result. An example of this is given in section 5.3 where we present a demo where we have encoded the edge normals.

### 4.4 Application Interfacing

Our implementation of CInDeR can be incorporated into an application. All that needs to be done is to include the class, and supply a projection function and a list holding the objects when the function for finding collisions is called. The reason for supplying a projection function every time the CInDeR algorithm is called, is that we want to be able change the projection function between frames. We may also want to examine the scene using more than one view in a single frame. This is especially useful if there is only a limited number of objects that one is interested in finding collisions with. For example when searching for collisions with one or more characters in a game, the perspective function can be adjusted to maximize the occupied screen area for those characters, without clipping possible colliding objects.

When the collision pairs are identified, CInDeR could be used to find other information about the collision by sending one pair at a time to a CInDeR function again. This time the encoded values could for example represent polygon-ids or edge normals. For an example where we have encoded edge normals see our demo in section 5.3.

Below is a skeleton with pseudo code showing how CInDeR could be used in an application.

```
cinderObject = new cinderObject()
projectionFunc = glProjectionMatrix(perspective)
objectList = sceneObjects.load()

while(true) {
    ...
    collisionPairs = cinderObject.findCollisions(objectList, projectionFunc)
    for (objectPair in collisionPairs){
        objectPair.collisionResponse()
    }
    ...
    scene.render(objectList)
}
```

CInDeR could be used as a broad phase algorithm before a CPU based narrow phased one. An issue that would have to be addressed in such a case is the approximations. In particular the loss of collisions due to edges being projected onto the same pixel.



## 5. Demos

We made some demos demonstrating different aspects of the algorithm, three of which are presented here. The first demo, Stairs, finds collisions between small solid spheres and the steps which are made up of boxes. The collision response is the same for all particles; they are reflected in the direction of the y-axis.

Snow Bunny has a particle system which makes up snow flakes. The particles are checked for collision with a bunny and the ground, and are frozen when they hit either.

For the third demo we wanted to identify the normal for the bunny in the contact point and calculate a response from that. We decided to code the edges in accordance to their normals instead of using edge ids. This gave us the necessary information to form a collision response directly and we used it to calculate a sphere-plane collision.

### 5.1 Stairs

This demo, see Figure 15, shows about 50 solid spheres bouncing down 11 solid “steps” (boxes). The collision points have been marked with purple rectangles. The spheres are shot out in a partly random direction at the top box. They are then affected by a gravity which increase their speed in the negative y-direction. When a ball hits a step it keeps its speed in the x-direction and z-direction, but the speed in y-direction is reflected and a bit dampened. For this setup the algorithm works fine and finds all concurrent collisions. When adding about 250 spheres there are sometimes around 30 simultaneous collisions. Some are masked out when they overlap and others that are close to each other in image space are lost in the mip-mapping. We only have a collision response for a sphere hitting a step and take no action when a sphere hits another sphere.



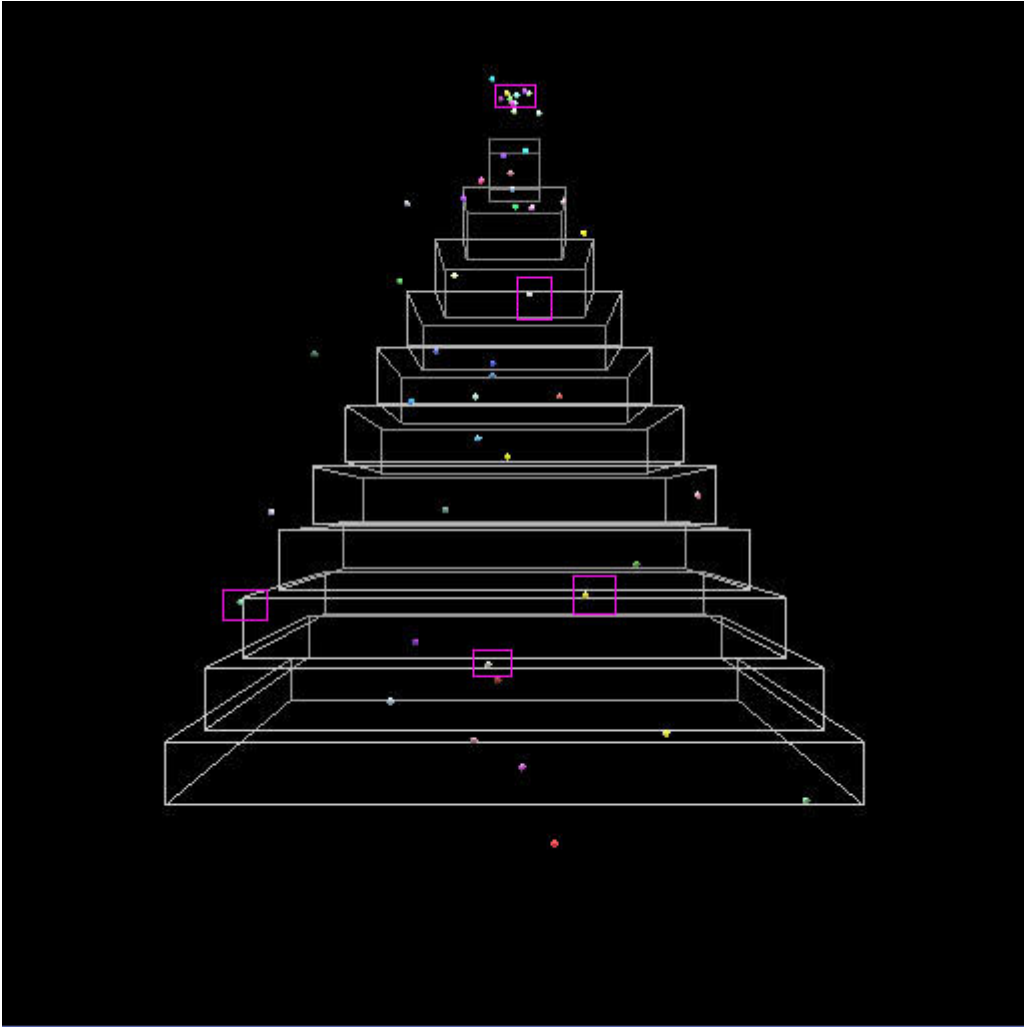


Figure 15: Stairs demo with the collisions marked out.

## 5.2 Snow Bunny

In this demo, see Figure 16, we let snowflakes fall down on a bunny. The snowflakes are built with 6 triangles and the bunny is the classic Stanford bunny, consisting of 8146 vertices. There is also a floor built of a solid box that the snowflakes can collide with. When a snowflake hits either the bunny or the floor it gets stuck. The stuck snowflake is also enlarged to make the interference more visible. The user can interact by rotating and moving the bunny. The goal with the demo is to show that CInDeR works good also for complex bodies with high polygon count. It is also interesting to investigate how the algorithm can be used for particle systems. The demo works fine although there are limitations in how fast we can spin the bunny and how fast the snow can fall. The reason is that if the objects move too fast, they might pass through the bunny or the floor between two frames, this as a result of the algorithm being image based.

The more objects that are stuck on the bunny and the floor, the lower the frame rate gets. When snowflakes hits the bunny or the floor they are moved to a separate list to make sure the algorithm doesn't analyze already stuck objects. The main problem in an algorithmic point of view is still that there can't be too many objects colliding in the same frame, especially if they are close to each other on the screen. The conclusion is that the algorithm works well also for complex models with a high polygon count, but is not well suited for particle systems. We were also reminded of one of the limitations of the algorithm, it doesn't work properly for non-solid objects. The bunny has a hole at the bottom, so in certain angles there can be strange result because there aren't the correct amount of front face and back face polygons.

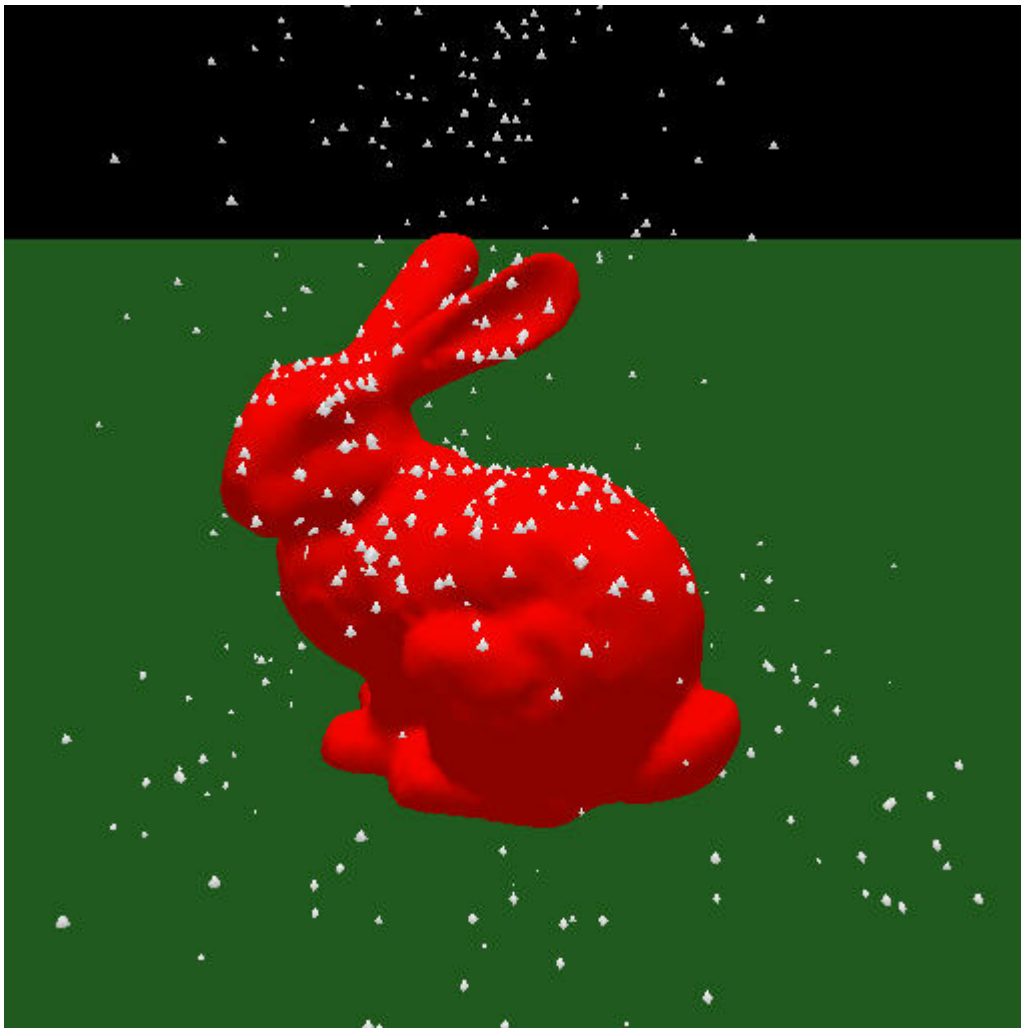


Figure 16: Bunny in the snow

## 5.3 Coloring the Edges

In the previous demos we have only received information about which objects collide in each frame. Usually this information is not enough to make a good collision response. In the case with the snowflakes that got stuck on the rabbit it was enough, since we could simply set the speed for the flake to zero. But if we have for instance larger spheres hitting the bunny, we might want them to bounce away in the correct direction. For such a computation we must have the normal for the surface in the contact point. The method suggested in the CInDeR paper [9] is to color each edge with a polygon-id instead of an object-id. This dramatically decreases the amount of objects that can be identified in a scene. If for example 8-bit color depth is used and 2 color channels are used to store one id, there can only be a maximum of 65535 polygons in the scene instead of equally many objects. From the identified polygon it's then easy to extract for example the normal, but since we are only interested in the normal at the contact point, we decided to skip polygon identification and color the edges for the bunny according to their normal instead. This is sufficient since we only have one bunny in our demo and no collision response when balls hit each other. The technique of polygon-id however follows the same principles. With our restrictions in mind we decided to encode the normal in the RGB channels and the sphere-id in the alpha channel.

### 5.3.1 Changes to the Algorithm

The first passes are color independent and the only difference is in the color coding during the identification step. Since we only get the normal of the bunny when it is the penetrating object, we don't detect a collision when a sphere is the penetrating object. The spheres are consequently only registered as penetrated objects and their id's are color coded to the alpha channel. With the bunny having such a dense mesh as it has, this will not be a big issue.

By doing these modifications the old filter function no longer works since it was based on two colliding objects having two channels each. The requirements for a found collision are that the alpha channel is non-zero and that at least one of the RGB channels are non-zero. If the alpha channel is zero and at least one of the RGB channels are non-zero then we have a bunny self-interference, and if all of the RGB channels are zero and the alpha channel is non-zero then we have a sphere-sphere collision or a sphere self-interference. The new filter simply deletes any found self-interference or sphere-sphere collisions according to the stated rules.

After the filtering function described above is applied, the collisions are extracted as before, yielding the colliding sphere-id in the alpha channel and the normal of the struck edge in the RGB channels. The collision response is then treated as a sphere-plane collision using the sphere velocity and the normal from the edge of the bunny.

### 5.3.2 Added Approximation

The source of errors added in this demo in regards to finding collisions, is the elimination of occurrences where the spheres are the penetrating objects. As a result of being a static interference detection algorithm the speed of the spheres and the bunny also play a role. If a sphere passes through the shell of the bunny between two consecutive frames a collision will not be registered. The effect of a sphere still being in interference with the bunny after a collision response can cause some strange effect like flickering bounces and odd directions. This can be avoided with issuing no response when the sphere is moving away from the collision plane. A screen shot of the scene can be viewed in Figure 17 below.

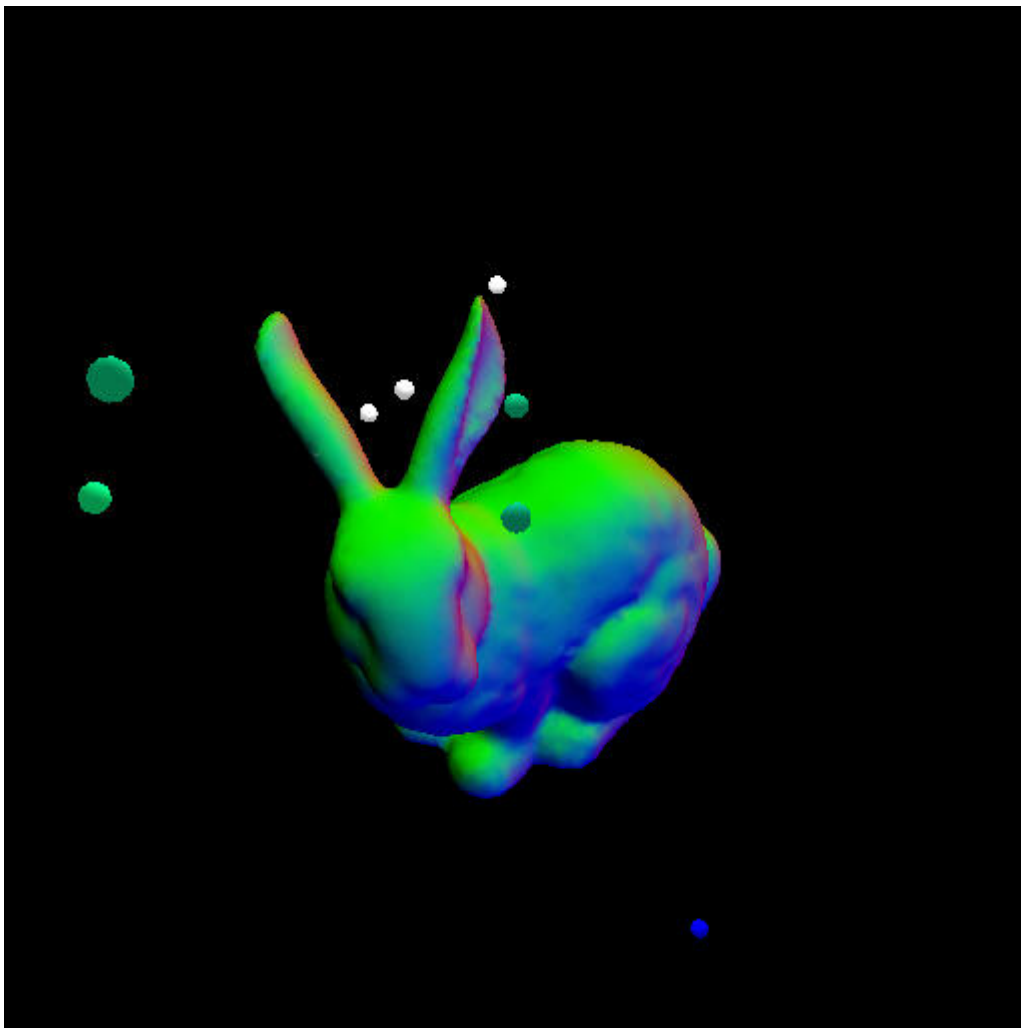


Figure 17: Bunny with colors according to the normals and colliding spheres with colors according to their collision points.



## 6. Result

In this paper it has been shown how the CInDeR algorithm can be used in practice for performing collision detection on the GPU. The algorithm was implemented as a stand-alone class which can be incorporated into an application, see section 4.4. The original version of CInDeR suffers from problems such as self-interferences and bandwidth limitations. These two problems are central to making the algorithm practically usable. Other less critical problems are the limitation of one collision per pixel and errors that can occur due to clipping, i.e. not preserving solid polyhedrons.

In the original paper about CInDeR [9] it is suggested that self-interferences could be avoided by offsetting the edges. This is however not an optimal solution as shown in section 4.3.1.1. We solved the problem of self-interferences by filtering them out using a Cg program before reading back the results. The drawback of filtering is that real collisions can still be overwritten by self-interferences before the filtering is applied. Instead we propose an alternate technique to avoid self-interferences altogether. By using an intermediate texture, self-interferences can be filtered using a simple fragment shader program.

A problem with image-space algorithms is the task of reading back the results to the CPU. In [9] it is stated that, in practice, one of the most time consuming parts of the algorithm is the act of retrieving the color buffer to the main memory and parsing through it one pixel at a time. It's also suggested in [9] that occlusion queries can be used to avoid sending data back when there is no collisions, or when there is only one collision pair to identify. The problem of identifying multiple collision pairs however still remain. Our approach was to reduce the data on the GPU before sending back the results. To accomplish this we used the technique of render-to-texture and implemented a mip-map algorithm in Cg. The mip-map algorithm reads the results which is stored in one texture and reduces it to a four times smaller one, for as many stages as are specified. The benefits are twofold; less data has to be sent back to the CPU and there is also less data left to parse.

Because the mip-map function is lossy, there is a tradeoff between speed and accuracy when choosing the number of mip-map stages. Occlusion queries could potentially be used to remove the information loss in our mip-map algorithm, see section 4.3.2.3. More sophisticated methods of processing the results on the GPU can be imagined. By reordering the results in a texture rather than simply mip-mapping, smaller collision results can be obtained without losing information.

The use of Cg opens new possibilities when formulating a collision result. We make use of this and examine the improvement of color coding arbitrary data. This is demonstrated by encoding the normals in the collision points and using these directly to calculate a collision response, thus eliminating the need to calculate the exact collision point on the CPU, and from there extract the normal.

## **6.1 Acknowledgments**

We would like to thank our supervisor Calle Lejdfors and assistant professor Lennart Ohlsson at the Department of Computer Science for helping us with this paper.

## 7. References

1. Computer Science and Telecommunications Board (CSTB), "Funding a Revolution: Government Support for Computing Research", 1999
2. Middle Tennessee State University, "History of Computer Graphics", <http://www.mtsu.edu/~csjudy/4250/history.htm>, 2000
3. CompHist.org, "A Brief History of Computer Graphics", [http://www.comphist.org/computing\\_history/new\\_page\\_6.htm](http://www.comphist.org/computing_history/new_page_6.htm), 2004
4. Wayne E. Carlson, "CGI Historical Timeline", <http://accad.osu.edu/~waynec/history/timeline.html>, 2004
5. oldcomputers.net, "Apple II", <http://oldcomputers.net/appleii.html>, 2004
6. Computer Hope.com, "Computer History - History from B.C. to today", <http://www.computerhope.com/history/index.htm>, 2005
7. ATI.com, "Radeon X850 Graphics Technology - Specifications", <http://www.ati.com/products/radeonx850/specs.html>, 2005
8. A1 Electronics.net, "Intel Pentium 4 600 & 9.73EE processors", [http://www.a1-electronics.net/Intel\\_Section/CPU/2005/Pentium4\\_600-series\\_Feb.shtml](http://www.a1-electronics.net/Intel_Section/CPU/2005/Pentium4_600-series_Feb.shtml), 2005
9. D. Knott and D. K. Pai, "CInDeR - Collision and Interference Detection in Real-time using Graphics Hardware", 2003
10. M. C. Lin and S. Gottschalk, "Collision detection between geometric models: a survey", 1998
11. P. Jiménez, F. Thomas and C. Torras, "3D Collision Detection: A Survey", 2000
12. R. Fernando and M. J. Kilgard, "The Cg Tutorial", 2003
13. T. Akenine-Möller and E. Haines, "Real-Time Rendering - Second Edition", 2002
14. M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification (Version 1.5)", 2003
15. R. S. Wright Jr. and B. Jipchak, "OpenGL SuperBible - Third Edition", 2004
16. <http://www.python.org>, "Python Programming Language"
17. <http://www.garagegames.com/blogs/20592/7164>, "Plan for Josh Williams", 2005



18. M. C. Lin and J. F. Canny, "A Fast Algorithm for Incremental Distance Calculation", IEEE Conference on Robotics and Automation, pages 1008-1014, 1991
19. E. G. Gilbert, D. W. Johnson and S. S. Keerthi, "A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space", IEEE Transactions on Robotics and Automation, volume 4, number 2, pages 193-203, April 1988
20. California Institute of Technology's Information Technology Services, "Collision Detection", <http://www.its.caltech.edu/~jiegao/collision-detection.html>, 2001
21. N. K. Govindaraju, S. Redon, M. C. Lin and D. Manocha, "CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware", 2003
22. G. Baciú, S.K. Wong, and H. Sun. "Recode: An image-based collision detection algorithm.", Proc. of Pacific Graphics, pages 497-512, 1998.
23. K. Myszkowski, O. G. Okunev, and T. L. Kunii, "Fast collision detection between complex solids using rasterizing graphics hardware." The Visual Computer, 11(9):497-512, 1995.
24. M. Shinya and M. C. Forgue. "Interference detection through rasterization.", The Journal of Visualization and Computer Animation, 2(4):131-134, 1991.
25. K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Fast computation of generalized voronoi diagrams using graphics hardware.", Proceedings of ACM SIGGRAPH, pages 277-286, 1999.
26. K. Hoff, A. Zaferakis, M. Lin, and D. Manocha, "Fast and simple 2d geometric proximity queries using graphics hardware.", Proc. of ACM Symposium on Interactive 3D Graphics, 2001.
27. Mark Harris, <http://www.markmark.net/misc/rendertexture.html>, "RenderTexture", 2003
28. R.O. Barr, "An efficient computational procedure for a generalized quadratic programming problem", SIAM J. Contr., vol.7, pp. 415-429, 1969
29. R. O. Barr and E. G. Gilbert, "Some efficient algorithms for a class of abstract optimization problems arising in optimal control", IEEE Trans. Automat. Contr., vol. AC-14, pp. 640-652, 1969
30. A. Watt and F. Policarpo, "3D Games - Real-time Rendering and Software Technology", Addison-Wesley, 2001

## Appendix A

Taken from [9]. We switched the lines 34 and 37, and 35 and 38.

### Algorithm 4 Identify both interfering objects

```
1: for all pixels do {clear colour, depth and stencil buffers}
2: Z=0, stencil=0, colour=(0, 0, 0)
3: end for
4: depth test = none
5: Enable depth update
6: stencil function = none
7: Enable colour update
8: for all objects do
9:   Draw edges blue {Pass #1}
10: end for
11: Disable colour update
12: Disable depth update
13: depth test = '<'
14: for all objects do
15:   cull mode = back-face
16:   stencil function = increment
17:   Draw polygons blue {Pass #2: add front-facing polygons}
18:   cull mode = front-face
19:   stencil function = decrement
20:   Draw polygons blue {Pass #3: subtract back-facing polygons}
21: end for
22: depth test = '='
23: Enable colour update
24: stencil test = '> 0'
25: stencil function = 'replace with 0'
26: for all objects do
27:   Draw edges blue {Pass #4: identify objects & reset stencil}
28: end for
29: stencil function = none
30: depth test = '<'
31: for all objects do
32:   stencil test = none
33:   Disable colour update
34:   cull mode = back-face
35:   stencil function = increment
36:   Draw polygons blue {Pass #5: add front-facing polygons}
37:   cull mode = front-face
38:   stencil function = decrement
39:   Draw polygons blue {Pass #6: subtract back-facing polygons}
40:   Enable colour update
41:   stencil test = ', 1'
42:   stencil function = 'replace with 0'
43:   Draw polygons blue {Pass #7 - identify object & reset stencil}
44: end for
45: for all pixels do {check for interference}
46:   if colour <> 0 then
47:     add colour channel 1 to list of interfering objects
48:     add colour channel 2 to list of interfering objects
49:   end if
50: end for
```