5-1-2012

Perceptually optimized real-time computer graphics

Jeffrey Smith

Perceptually Optimized Real-Time Computer Graphics

by

Jeffrey D. Smith

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Supervised by

Assistant Professor Dr. Reynold Bailey
Department of Computer Science
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2012

Approved by:

Dr. Reynold Bailey, Assistant Professor
Thesis Advisor, Department of Computer Science

Dr. Roy Melton, Senior Lecturer
Committee Member, Department of Computer Engineering

Dr. Muhammad Shaaban, Associate Professor
Committee Member, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Perceptually Optimized Real-Time Computer Graphics

I, Jeffrey D. Smith, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Jeffrey D. Smith

Date

Acknowledgments

I would like to express my sincere gratitude to Dr. Reynold Bailey for introducing me to the field of computer graphics and serving as my primary thesis advisor. I would also like to thank my committee members Dr. Roy Melton and Dr. Muhammad Shaaban for providing me with their professional guidance and support throughout the development of this thesis, as well as Anoop Thomas, Srinivas Sridharan and Sean Xu for their help in bringing me up to speed with GPGPU computing and eye-tracking methodology. Finally, I would like to thank the National Science Foundation for funding this work.

Abstract

Perceptually Optimized
Real-Time Computer Graphics

Jeffrey D. Smith

Supervising Professor: Dr. Reynold Bailey

Perceptual optimization, the application of human visual perception models to remove imperceptible components in a graphics system, has been proven effective in achieving significant computational speedup. Previous implementations of this technique have focused on spatial level of detail reduction, which typically results in noticeable degradation of image quality. This thesis introduces refresh rate modulation (RRM), a novel perceptual optimization technique that produces better performance enhancement while more effectively preserving image quality and resolving static scene elements in full detail.

In order to demonstrate the effectiveness of this technique, a graphics framework has been developed that interfaces with eye tracking hardware to take advantage of user fixation data in real-time. Central to the framework is a high-performance GPGPU ray-tracing engine written in OpenCL. RRM reduces the frequency with which pixels outside of the foveal region are updated by the ray-tracer. A persistent pixel buffer is maintained such that peripheral data from previous frames provides context for the foveal image in the current frame. Traditional optimization techniques have also been incorporated into the ray-tracer for improved performance.

Applying the RRM technique to the ray-tracing engine results in a speedup of 2.27 (252 fps vs. 111 fps at 1080p) for the classic Whitted scene with reflection and transmission enabled. A speedup of 3.41 (140 fps vs. 41 fps at 1080p) is observed for a high-polygon scene that depicts the Stanford Bunny. A small pilot study indicates that RRM achieves these results with minimal impact to perceived image quality.

A secondary investigation is conducted regarding the performance benefits of increasing physics engine error tolerance for bounding volume hierarchy based collision detection when the scene elements involved are in the user's periphery. The open-source Bullet Physics Library was used to add accurate collision detection to the full resolution ray-tracing engine. For a scene with a static high-polygon model and 50 moving spheres, a speedup of 1.8 was observed for physics calculations. The development and integration of this subsystem demonstrates the extensibility of the graphics framework.

Contents

List of Figures

Chapter 1

Introduction

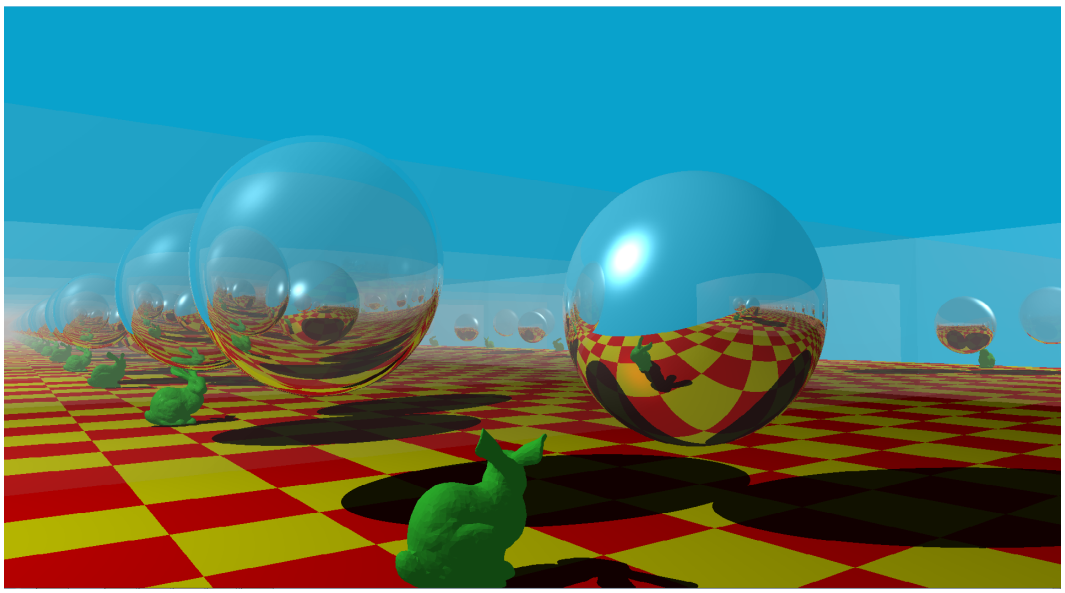


Figure 1.1: 1080p resolution ray-traced image generated using this framework.

Recent advances in consumer level parallel processing hardware have led to the feasibility of generating realistic computer graphics images at interactive rates. However, even with high-end hardware, computationally demanding rendering solutions such as ray-tracing must be heavily optimized to run in real-time.

A number of acceleration techniques have been proven effective in reducing rendering time for ray-tracing applications. These include the use of spatial data structures such as k-d trees [32] or bounding volume hierarchies [10]. Despite these advances, computational resources are still dedicated to generating fine detail outside of the viewer's high acuity foveal

region. These resources are wasted as the presence of such details does not impact the perceived quality of the scene due to reduced acuity in the peripheral region of the field of view. Raj et al. [22] noted however, that peripheral vision is not simply a blurred version of foveal vision; hence the traditional perceptual optimization approach of reducing spatial detail in the periphery still results in a noticeable reduction in image quality.

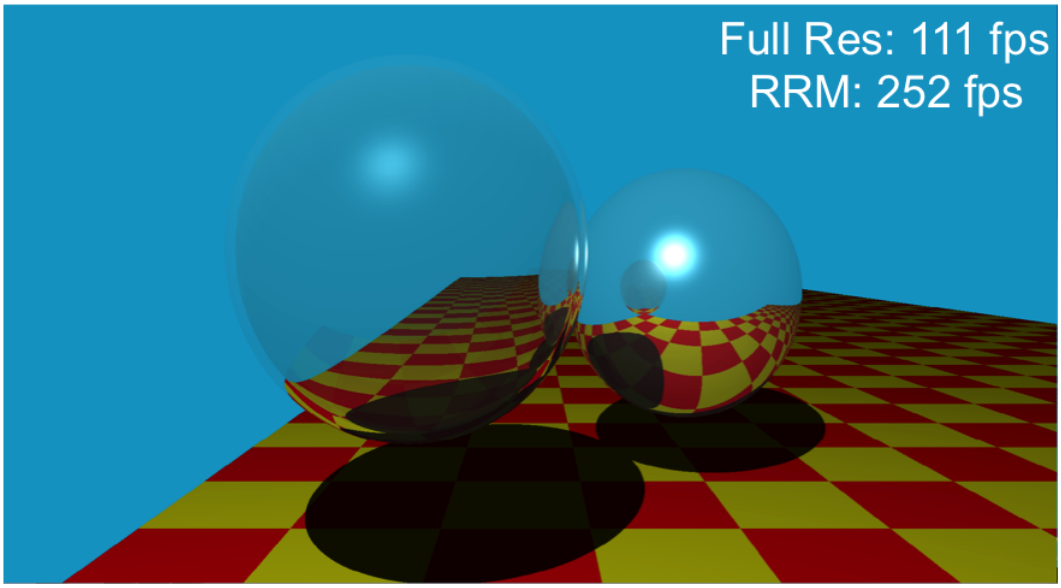


Figure 1.2: This framework renders the classic Whitted scene in 1080p at 111 fps in full resolution and 252 fps when RRM is enabled.

This thesis introduces refresh rate modulation (RRM), a novel perceptual optimization technique that produces better performance enhancement than spatial degradation techniques while more effectively preserving perceived image quality (Figure 1.2). Similar to variable resolution approaches, RRM partitions the display area into two subregions that correspond to the foveal and peripheral portions of the user's field of view. However, instead of varying sampling frequency, RRM adjusts the rate at which pixels are updated by the ray-tracer. The foveal region is updated once per frame, and therefore shows the scene in full detail at all times. Pixels in the peripheral region are refreshed once every N frames, where N can be adjusted to strike a balance between performance and perceived output quality.

The result is a subtle fragmentation effect outside of the foveal region

that does not decrease the perceived quality of the overall image, but significantly increases performance. If the scene remains still for N or more frames, all peripheral pixels are refreshed and a full-detail image of the entire viewing area is rendered (the result of this behavior is shown in Figure 1.1).

Within the framework, physics calculations may also be optimized through the use of real-time perceptual data. While the center of the field of view is able to detect errors in physical phenomena with high accuracy, the periphery is less well-equipped to do so [20]. This means that collision error tolerances can be significantly increased in regions outside of the fovea without reducing the perceived quality of motion (so long as penetrative errors are avoided). Many physics engines utilize acceleration structures for polygonal meshes that result in a series of successive calculations for collision detection between two objects. If the collision algorithm is modified to return a collision several layers earlier, computation for collisions with the mesh terminate early, and a computational speedup occurs.

The remainder of this document is organized as follows: background and related work are presented in Chapter 2. The design of the perceptually optimized rendering framework is described in Chapter 3. Performance results and a pilot study to gauge the perceptibility of the novel refresh rate modulation technique are presented in Chapter 4. In Chapter 5, the paper concludes with a summary of the contributions and potential avenues of future research.

Chapter 2

Background and Related Work

2.1 Human Visual Perception

The human visual system is made up of a complex set of sensory and processing organs that interact to produce sight. The visual process begins when light enters the eye through the pupil after bouncing off various surfaces in the environment. The lens focuses light and directs it toward the back of the eye. A muscle surrounding the lens expands and contracts, changing the shape of the lens to match environmental conditions and enabling variable focus. Light then travels through the vitreous humor, a transparent gel that fills the eye, to the retina on the rear wall.

The human retina contains a large number of interconnected receptor cells that intercept incoming photons and output electrical signals to the visual cortex. Cone cells provide color vision at high illumination levels, and are responsible for detail-oriented visual tasks. The distribution of cone cells across the retina is nonuniform (see Figure 2.1). Millions of these cells are packed into the macula, a small region at the center of the retina. The highest concentration of cone cells is in the center of the macula, the fovea centralis. While the fovea centralis accounts for less than one percent of total retinal area, approximately 50% of the information transmitted to the brain is generated by this region [24].

Nonuniform cone distribution results in two distinct regions within the field of view: the central, high acuity fovea, and the outer, low acuity periphery. Foveal vision subtends 1°–5° of visual angle (see Figure 2.2), which means that only a tiny portion of the field of view is perceived in high-detail

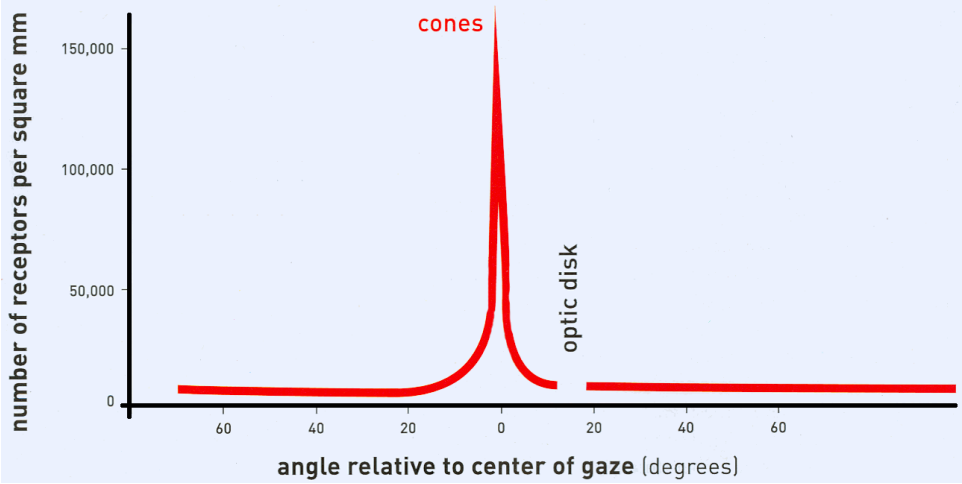


Figure 2.1: Distribution of cones in the retina. Adapted from [14]. Cones are densely packed in the center of gaze (fovea) and the density of cones falls off rapidly as angle from the center of gaze increases. The distribution of cones directly affects visual acuity. Visual acuity is highest in the center of gaze and falls off rapidly as angle from the center of gaze increases.

at any given time. For reference, the average person's thumbnail is equivalent to a visual angle of 1.5°–2° when held at arm's length. In computer graphics terms, only three percent of a 21-inch computer monitor viewed at 60 centimeters lies within this region [5]. The visual system reorients the eye an average of three times per second via saccadic movement, and integrates the information gathered at each fixation point to create a composite perceived image that seems to be in full detail. Signals produced by the retina are transmitted via the optic nerve to the visual cortex, the region of the brain responsible for image processing [17].

Figure 2.2: Visual angle describes the size of objects in the viewing region [5].

2.2 Eye Tracking

The method most commonly used by current eye-tracking hardware is video-based infrared oculography. A light source emits infrared light toward the subject, which creates a series of four reflections on the eye, one each from the front and back of the cornea and lens. While all four reflections can be used to generate extremely precise fixation data, typically only the reflection from the front of the cornea is measured [5].

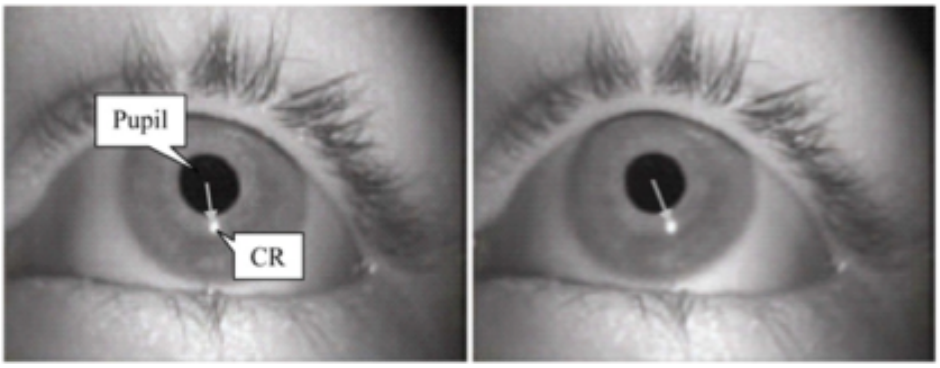


Figure 2.3: Monitoring distance between pupil center and corneal reflection yields viable fixation data [12].

The position of the pupil center is calculated by applying simple image processing techniques to each video frame. Because the difference between the pupil center and the corneal reflection changes with eye rotation but remains relatively constant with normal head movements, the rotational position of the eye can be accurately determined from the distance from the corneal reflection to the pupil center and the angle between them (Figure 2.3). After an initial calibration step that relates eye rotation to screen coordinates, the onscreen location of the user's fixation can be reported in real-time.

2.3 Ray Tracing Algorithm

2.3.1 Overview

Ray tracing is a physically based computer graphics technique that generates images by shooting rays that begin at the eye point and travel through a viewing plane and into the scene (Figure 2.4). The basic algorithm creates one ray for each onscreen pixel. If a ray intersects an object in the scene, the associated pixel takes on the color of the object at the point of intersection. Object color is determined using an illumination model that simulates diffuse and specular reflections. Shadows are handled by spawning an additional ray towards each light source in the scene from an intersection point. If any object is between the light source and the point of intersection, that point is in shadow. If the object is reflective or transmissive, secondary rays are spawned recursively and contribute to the final color of the pixel.

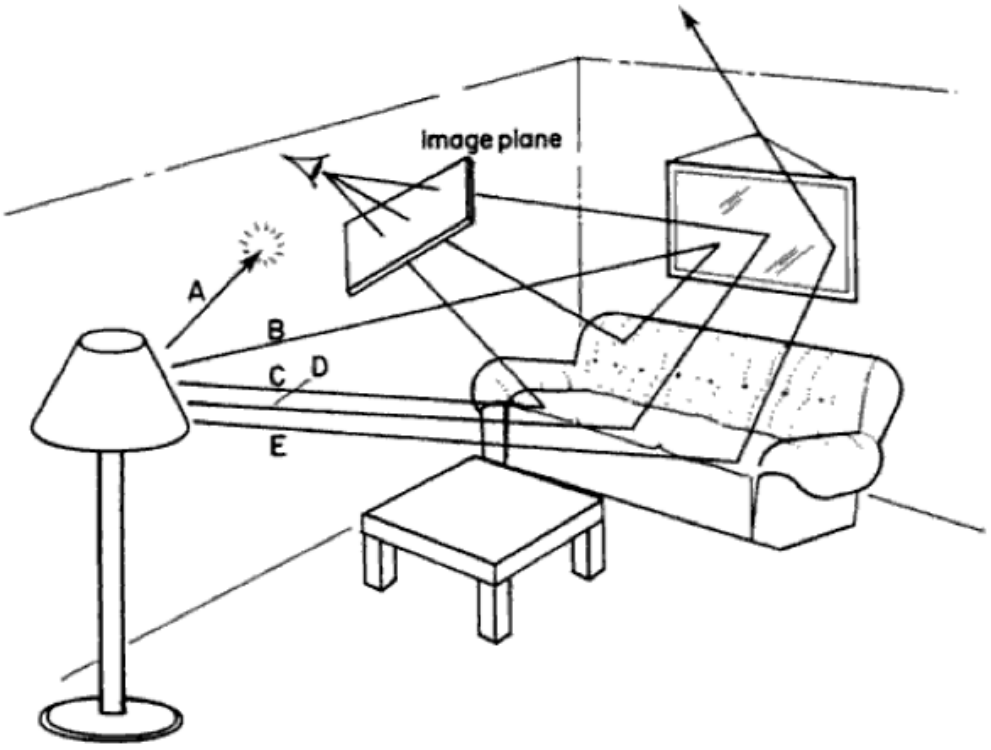


Figure 2.4: Rays are generated at the eye point and traced through the viewing plane and into the scene [9].

2.3.2 Ray Definition

A ray is an infinite straight line that is defined by an origin point P_0 and a unit vector direction D. For a viewing model with no antialiasing, one ray is spawned for every onscreen pixel, where the screen is represented by the view plane (Figure 2.5). The origin for all primary rays is the eye point, and the direction is calculated by drawing a line from the eye point to the center of the associated pixel and normalizing the resultant vector. For each frame, every ray must be tested for intersection with all scene objects to determine which color should be assigned to each onscreen pixel.

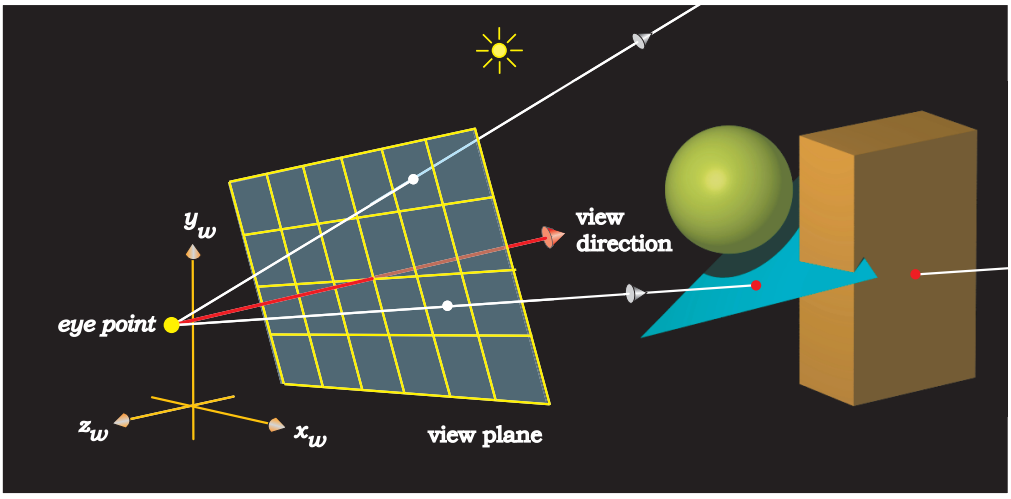


Figure 2.5: One ray is generated for each onscreen pixel [29].

2.3.3 Ray-Object Intersection

Introduction

Ray-object intersection calculations are performed using formulas that are derived by substituting the parametric representation of a ray into geometric object equations. The parametric form of a ray is shown in (2.3). The quantity t represents the distance of an intersection from the origin of the ray at point P_0 (2.1) in the normalized direction vector D (2.2).

$$P_0 = (x_0, y_0, z_0) \tag{2.1}$$

$$D = (dx, dy, dz) \tag{2.2}$$

$$R = P_0 + tD \tag{2.3}$$

Splitting (2.3) into x, y, and z components yields the expressions shown in (2.4) through (2.6). This form is used to derive intersection formulas.

$$x = x_0 + tdx \tag{2.4}$$

$$y = y_0 + tdy \tag{2.5}$$

$$z = z_0 + tdz \tag{2.6}$$

Ray-Sphere Intersection

The simplest geometric object to intersect with a ray is the sphere. Sphere objects are used to encapsulate other objects for some acceleration techniques, as well as to render perfectly smooth parametric spheres that are not composed of polygons. (2.7) shows the basic equation for a point (x_s, y_s, z_s) on a sphere with center (x_c, y_c, z_c) and radius r.

$$(x_s - x_c)^2 + (y_s - y_c)^2 + (z_s - z_c)^2 = r^2 \tag{2.7}$$

Substituting the parametric expressions from (2.4) through (2.6) for x_s, y_s and z_s in this equation yields (2.8) (where A, B, and C are given in (2.9) through (2.11).

$$At^2 + Bt + c = 0 \tag{2.8}$$

$$A = dx^2 + dy^2 + dz^2 \tag{2.9}$$

$$B = 2(dx(x_0 - x_c) + dy(y_0 - y_c) + dz(z_0 - z_c))) \tag{2.10}$$

$$C = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2 \tag{2.11}$$

Solving (2.8) for t produces (2.12), the discriminant of which can be used to classify the intersection between the ray and sphere.

$$t = \frac{-B \pm \sqrt{B^2 - 4C}}{2} \tag{2.12}$$

Figure 2.6 illustrates this classification process, which avoids an imaginary result for t and bypasses several floating point calculations in the case of no intersection. If $d = B^2 - 4C$ is negative, there are no intersections and intersection computation can move on to the next object. If d is equal to 0, there is guaranteed to be only a single intersection. If d is greater than 0, two intersections exist, the closer of which should be used as it obscures the farther intersection. Substituting the solution for t from (2.12) into (2.3) yields the point of intersection $R = (x_i, y_i, z_i)$.

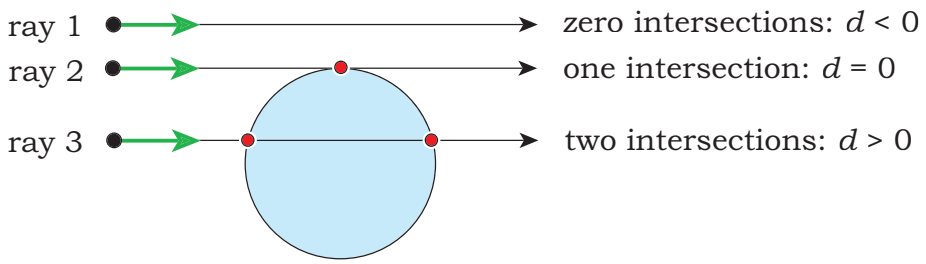


Figure 2.6: Ray-sphere intersection is characterized by the discriminant of (2.12) [29].

Ray-Triangle Intersection

Scene objects that are not parametrically defined are made up of groups of adjoining triangles. The ray-triangle intersection test begins by determining whether the ray passes through the plane defined by the triangle. (2.13) is used to find the distance from the ray origin to the point of intersection, where n is a vector normal to the triangle and p_1 is a triangle vertex. If $n \cdot D$ equals 0, the ray is parallel to the plane and there is no intersection. If t is negative, the intersection occurs behind the eye point and is ignored (Figure 2.7). Otherwise, there is a valid intersection.

$$t = \frac{-(n \cdot P_0 - n \cdot p_1)}{n \cdot D} \tag{2.13}$$

Once ray-plane intersection has been confirmed, the intersection point must be checked to determine if it is within the vertices of the triangle. This is accomplished by forming a vector between the intersection point and each vertex, and then summing the angles between each vector. If the

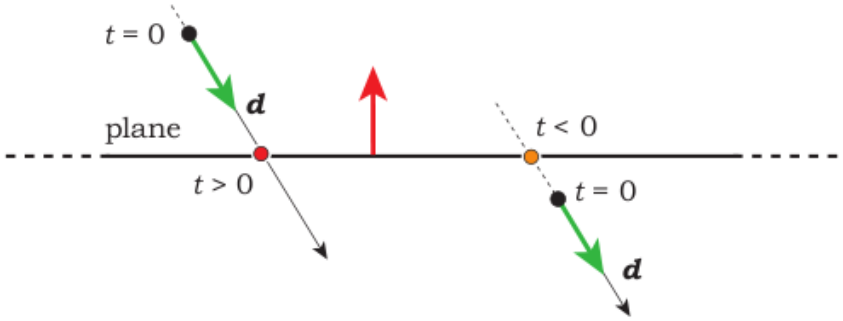


Figure 2.7: Ray-triangle intersection begins by intersecting the plane in which the triangle lies. Adapted from [29].

sum is $360\,^{\circ}$, the point is inside the triangle and the ray intersection is valid. Otherwise, there is no intersection. Quadrilateral polygons are processed in a similar manner, with the fourth vertex added to the final angle calculation.

2.3.4 Illumination and Shading Models

Object color is determined using a combination of illumination and shading models. An illumination model describes the reflective characteristics of a surface, and impacts how light in the scene will interact with it. Illumination models serve as an approximation to the Bidirectional Reflectance Distribution Function (BRDF, (2.14)), which relates reflected radiance (light emitted from a surface) in one direction ω_0 to irradiance (light incident on a surface) centered in another direction ω_i. Figure 2.8 illustrates the geometry involved in the BRDF.

$$BRDF = f_r(\phi_i, \theta_i, \phi_r, \theta_r) \tag{2.14}$$

Since true global illumination techniques that accurately model the flow of light in a scene are very computationally expensive, illumination models that are suitable for real-time applications divide light into three components: ambient, diffuse, and specular. The ambient component models light that is spread uniformly throughout the environment. It is a crude (but much faster) substitute for physically accurate diffuse interreflection, and adds some color to objects that are shadowed from all scene light sources.

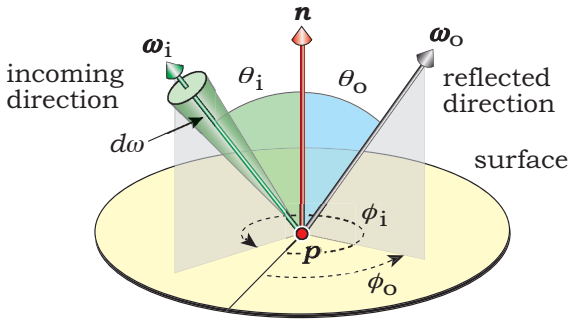


Figure 2.8: The BRDF relates reflected radiance to irradiance [29].

The diffuse component represents light that is scattered equally in all directions from the point of contact. The specular component represents light that is perfectly reflected from the object to the viewer, and adds specular highlights. Figure 2.9 illustrates the effect of each of the three components on the final image.

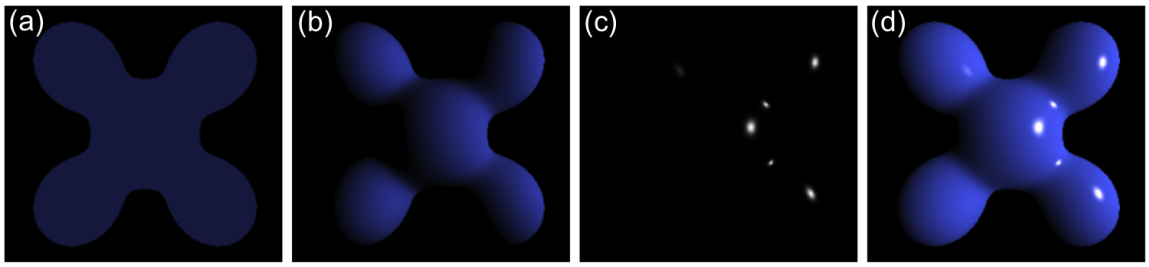


Figure 2.9: Phong shading adds ambient, diffuse and specular light components to produce a realistic result [36]. (a) Ambient. (b) Diffuse. (c) Specular. (d) Combined.

The shading model uses the information generated by the illumination model to determine object color at each point. (2.15) shows the Phong shading formula, which uses the vectors shown in Figure 2.10 along with parameters for ambient (k_a), diffuse, (k_d) and specular (k_s) response. Light source (L_i) and object colors (L_a) are also incorporated.

$$L(V) = k_a L_a + k_d \Sigma_i L_i (l_i \cdot n) + k_s \Sigma_i L_i (r_i \cdot w_0)^{k_e} \tag{2.15}$$

If a bitmap or procedural object texture is desired, the color for the point of intersection should be used for L_a in the Phong equation. The second and

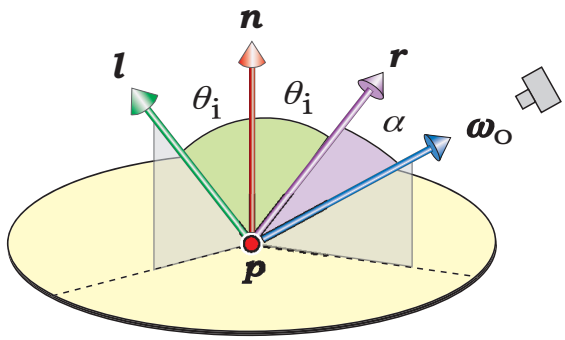


Figure 2.10: Phong shading requires unit vectors for light source (l), surface normal (n), reflection (r) and viewing direction (w_0) [29].

third terms in the Phong equation include a summation to account for multiple light sources with different positions and colors. The light source vector l and the reflection vector r will be different for each light source, while the viewing direction vector w_0 and the normal vector n remain unchanged through the summation.

Shadows are produced by removing the diffuse and specular color components for points that are blocked from all light sources. After a valid intersection point is detected via the ray-object intersection tests, a shadow-ray is spawned towards each light source, and the ray-object intersections are repeated. If any object intersection that has a positive t value and is also not beyond the light source is found, the point is shadowed from that light source. Figure 2.11 illustrates shadow-ray generation.

2.3.5 Reflection and Transmission

For surfaces that are reflective or transmissive, additional color data must be added at the point of intersection to produce a realistic result. As shown in (2.16), these color data come in the form of reflection and transmission terms that are added to the result from the local illumination model. Objects have associated constants of reflection (k_r) and transmission (k_t) to indicate the desired intensity of each effect. Color data are acquired by spawning secondary rays at the point of intersection.

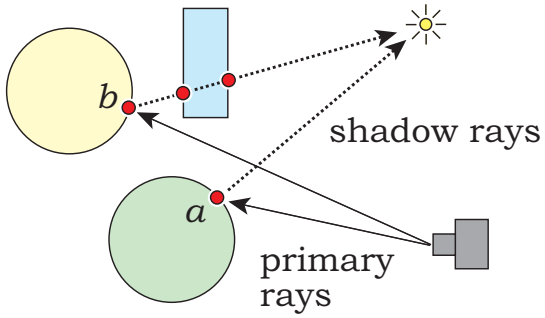


Figure 2.11: Regions of shadow are found by spawning a shadow-ray toward each light source in the scene. If the ray intersects any object, the point is in shadow [29].

$$I = I_{local} + k_r I_{reflected} + k_t I_{transmitted} \tag{2.16}$$

A reflection ray is spawned when a ray intersects an object with $k_r > 0$ (Figure 2.12). The direction of the reflected ray r is given by (2.17), with the origin at the point of intersection p. r_0 is a unit vector indicating the direction from the viewing point to p.

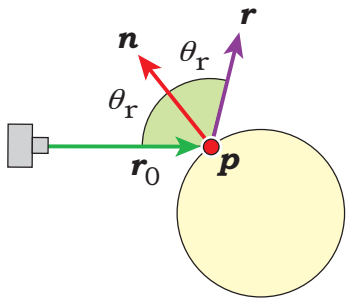


Figure 2.12: Reflected ray r is spawned when ray r_0 intersects point p on a surface that has a non-zero constant of reflection and surface normal n [29].

$$r = r_0 - 2(\frac{r_0 \cdot n}{||n^2||})n \tag{2.17}$$

If a reflection ray intersects another reflective surface, a second reflection ray is generated and also contributes to the final pixel color (Figure 2.13). The constant of reflection is compounded at each reflection step, so subsequent reflections grow fainter. The number of reflections allowed must be

limited to prevent performance issues for scenes with many reflective surfaces, as only the first few reflections for a given surface have a perceptible impact on the image.

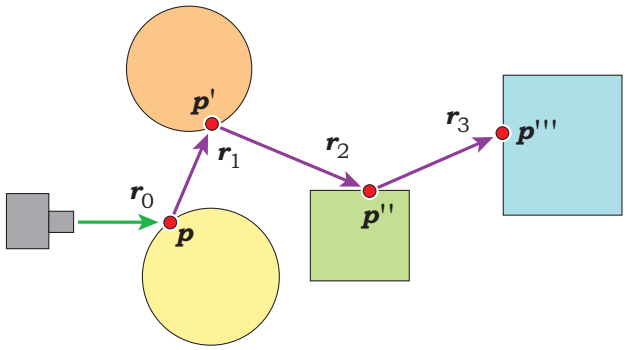


Figure 2.13: If a reflection ray intersects an object that is also reflective, another reflection ray is spawned at the point of intersection [29].

A transmission ray is spawned when a primary or secondary ray intersects a transparent object. Figure 2.14 illustrates the geometry of ray transmission. Snell's Law is used to derive (2.18), which gives the direction of transmission ray t (transmission rays also begin at the point of intersection p). $n_{it} = \frac{n_{out}}{n_{in}}$ is the ratio of indices of refraction of the outer and inner media [29]. Commonly depicted transmissive materials include glass ($k_t = 0.95$) and air ($k_t = 1.0$).

$$t = n_{it}w_0 + (n_{it}(-w_0 \cdot n) - \sqrt{1 + (n_{it}^2((-w_0 \cdot n)^2 - 1))})n \qquad (2.18)$$

2.3.6 Ray Tracing vs. Rasterization

The ray tracing algorithm is not widely used in real-time applications due to the large computational overhead that it incurs. However, since it models the physical behavior of light, it is able to handle a number of common but complex rendering situations in a simple and intuitive way. In most cases, it produces better visual results than traditional rasterization while more closely approximating the actual behavior of real-world phenomena [29].

Reflection is one area in which rasterization methods are particularly

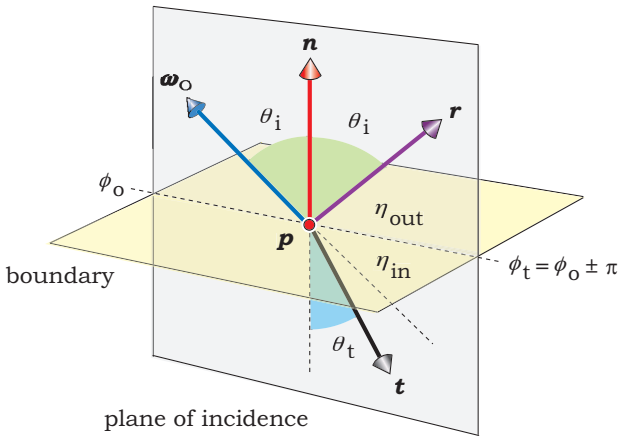


Figure 2.14: Transmission ray t is spawned when ray w_0 intersects point p on a surface that has a non-zero constant of transmission [29].

weak. Since the algorithm is not physically-based, environment maps must be used to approximate true reflection. While maps can be applied to curved objects, the map itself must be derived from a flat projection of the scene. One popular technique, cube projection, uses six virtual cameras to build projections for the walls, floor and ceiling of the cube (Figure 2.15). Environment mapping produces visually acceptable results, but it is strictly inferior to ray-traced reflection [2].

Transparency is another area in which ray tracing excels and rasterization falls short. Blending must be used to mimic true transparency for rasterization. To achieve this, an opacity value is assigned to fragments as the associated polygons are rendered into the frame buffer. As shown in Figure 2.16, pixel color is calculated by adding together overlapping transparent objects, taking into account the opacity of each fragment and their arrangement. Creating realistic refraction effects is not possible with this technique.

The ray tracing algorithm is also better suited to handle a number of global illumination techniques and other effects. Photon mapping and radiosity can both be achieved using the standard ray-tracing framework. Subsurface scattering can also be implemented easily with ray structures, and

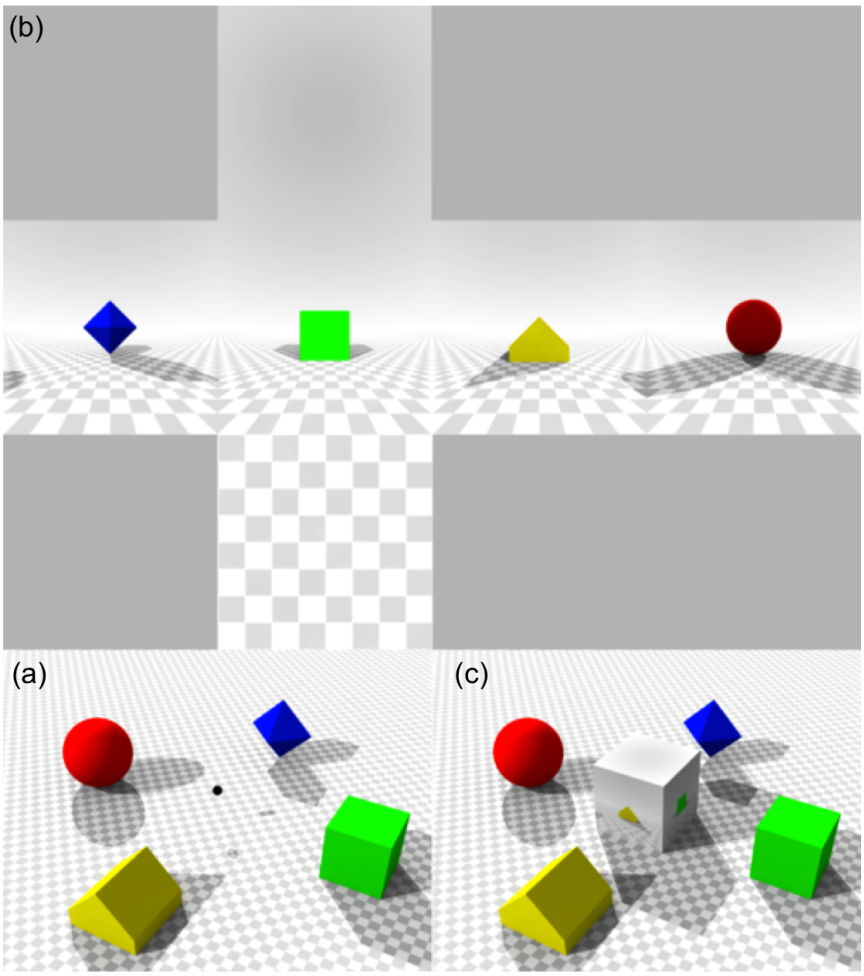


Figure 2.15: Cube mapping. Adapted from [35]. (a) Sample scene with desired cube map center marked with a black dot. (b) Cube mapping as seen from viewpoint. (c) Cube map superimposed on original scene.

supersampling is simply a matter of spawning more rays than there are pixels. Overall, ray tracing is a superior graphics solution, but the high overhead that it incurs must be overcome with acceleration techniques and high-performance hardware to produce a viable real-time system.

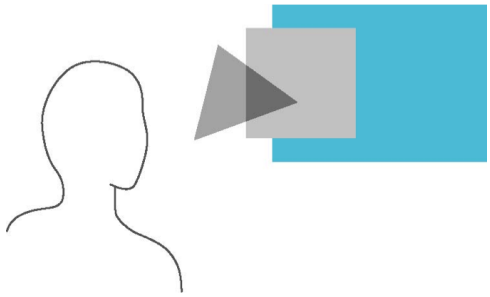


Figure 2.16: Blending. An opacity value is assigned to each polygon, and overlapping fragments are added together [2].

2.4 Ray-Tracing Acceleration Structures

During the development and implementation of the ray tracing algorithm, Whitted noted that approximately 75% of rendering time for simple scenes was allocated to ray-object intersections [31]. This percentage grows larger for more complex scenes, where the number of intersections per ray increases. Several techniques reduce the average number of intersection calculations per ray by organizing scene objects in such a way that each intersection test removes many objects from consideration. Such techniques include the bounding volume hierarchy, the uniform grid, the binary space partitioning tree and the k-d tree. Each technique exhibits different strengths and is best-suited to unique scene conditions and system design considerations.

2.4.1 Bounding Volume Hierarchy

In order to decrease the number of intersection calculations associated with each ray, the objects in the scene can be placed in simple bounding volumes. These volumes tend to have faster intersection algorithms, (e.g. a cube bounding volume for polygonal objects, Figure 2.17), and can contain multiple objects.

These volumes are organized into a hierarchical structure by repeatedly grouping together two or more bounding volumes at each level and surrounding the group with another larger bounding volume (Figure 2.18). The resulting tree is then searched recursively, using the idea that a ray cannot

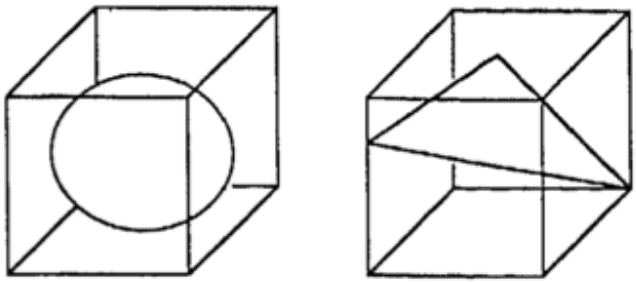


Figure 2.17: Objects surrounded by rectangular bounding volumes [10].

possibly intersect child volumes if it does not intersect the larger parent. At each level, up to half of the remaining nodes are eliminated from consideration. Using a bounding volume hierarchy can produce very good results, with a best case performance of $O(log\ n)$ intersection calculations per ray [10].

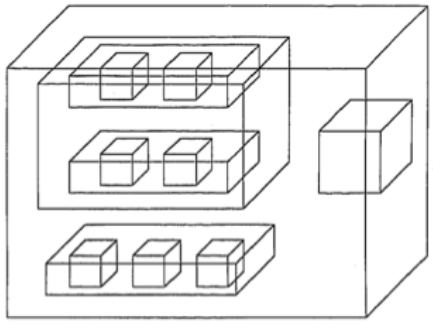


Figure 2.18: Bounding volume hierarchy [10].

For static scenes, performance of the construction algorithm is not extremely important, as the hierarchy can be precomputed and then loaded at runtime. However, achieving a tree with near-optimal traversal efficiency is vital and can easily cause a 50x improvement in traversal performance over a suboptimal tree [31].

Hierarchy construction algorithms generally iterate over all objects in the scene and evaluate the traversal cost of placing the current object in a small set of available locations in the tree. The least costly location is selected, and the algorithm moves to the next object with the updated tree (Figure 2.19).

Alternatively, the entire tree can be built by randomly pairing all nodes

at each level. The tree is scored based on average traversal cost and stored in a buffer or written to disk. Another random tree is built and compared to the previous tree - if its score is higher, it replaces the current best tree. This process is repeated until either a traversal-cost tolerance is reached, or a set number of iterations has occurred. It can result in a close to optimal tree in much less time than brute force algorithms [31].

When designing a high-performance ray tracer, multiple acceleration techniques must be put in place. Doing so creates interconnected requirements that affect the implementation of all techniques. The most important consideration is the ability to be easily represented to and computed by a GPU that operates under the current stream processor paradigm. The languages used to work with current GPUs generally conform to the C99 standard, which means that object oriented techniques are not available. In addition, recursive functions are not allowed. This means that the quality of an acceleration technique must take into consideration the unconventional issues that porting to a stream processor introduces.

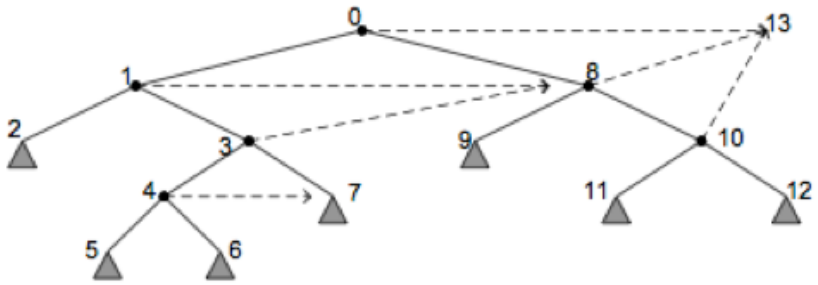


Figure 2.19: Bounding volume hierarchy with escape indices [31].

Using the bounding volume hierarchy for computational speedup on a GPU device leads to two issues: the tree must be traversable without the use of a stack, and the hierarchy must be able to be passed to the kernel in an efficient way [31]. One effective means of representing the bounding volume hierarchy is illustrated in Figure 2.19. The indices of the hierarchy nodes are stored in order of left to right traversal. An escape index is stored that indicates to which node to move for each node if the intersection test for the current node returns false (Figure 2.20). This scheme places most of the work involved in traversing the tree on the side of the hierarchy construction

algorithm, and in turn results in a very simple traversal algorithm for the GPU kernel.

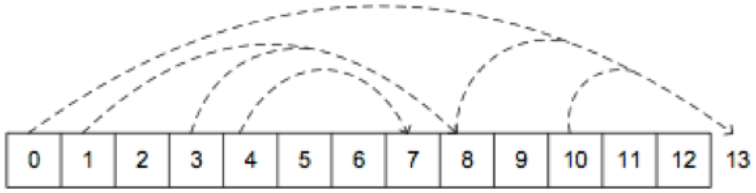


Figure 2.20: Bounding volume hierarchy encoding data structure [31].

2.4.2 Uniform Grid

The uniform grid method splits the populated region of the scene into a regular three-dimensional grid with an arbitrary cell size. Cell size significantly affects performance, and should be chosen based on the scale and distribution of the objects in the scene. A preprocessing algorithm assigns all objects to grid cells based on scene position [26].

When rendering takes place, the cells of the grid are traversed in order of their intersection with the current ray. The objects located within the first cell are checked for intersection with the ray. If an intersection is found, the remaining objects in the cell are tested and the algorithm returns that closest object in that cell. If an intersection is not found, the next cell is checked. This process continues until either a valid intersection is found or all populated cells have been depleted (Figure 2.21).

Scene contents has a much larger impact on performance for this algorithm than for the binary space partitioning tree or bounding volume hierarchy techniques. If there are many similarly sized objects spread evenly throughout the scene, this technique is very effective. If the objects vary widely in scale and are located in clusters throughout the scene, the performance of the algorithm drops significantly. However, the preprocessing step is not as computationally intensive for the uniform grid approach, so it can be more easily applied to dynamic scenes.

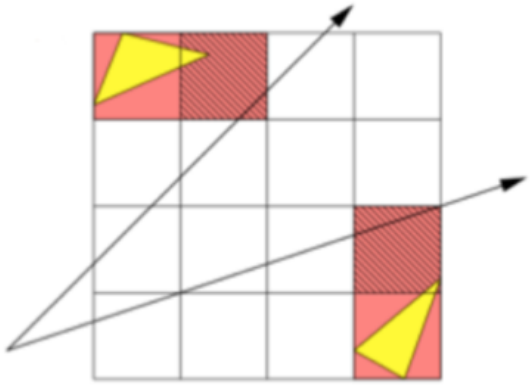


Figure 2.21: Ray-object intersection is guided by a regular three-dimensional grid. Each cell that the ray passes through is checked in order for intersecting objects. Traversal ends when a valid intersection is found.

2.4.3 Binary Space Partitioning Tree

The binary space-partitioning tree method involves recursively splitting the scene into two regions with an arbitrary plane. The goal of the splitting step is to create two equal groups of objects on either side of the plane. Once the two groups are created, a plane is calculated to split each into two new groups, with a subset of the objects located in each group. This process is repeated either until each group has a sufficiently small number of objects and is considered a leaf node, or until there is no plane that will cleanly split any of the current groups [6].

Figure 2.22 shows the binary space partitioning tree construction process. Group A, which contains all the polygons within the scene, is divided into groups B and C with an optimal splitting plane that results in the most even grouping possible. Group B is then recursively divided into groups E and D, and then group D is divided into groups G and F. At the end of this example, groups G and F are leaf nodes and contain two and three polygons, respectively. Groups E and C remain unfinished.

While binary space partitioning trees with arbitrary splitting planes successfully produce a speedup for ray tracing applications, the computational cost of intersecting rays with off-axis planes is relatively high. This cost can be significantly reduced by selecting only planes that lie on one of the coordinate axes for a three-dimensional scene. The cost of building the tree

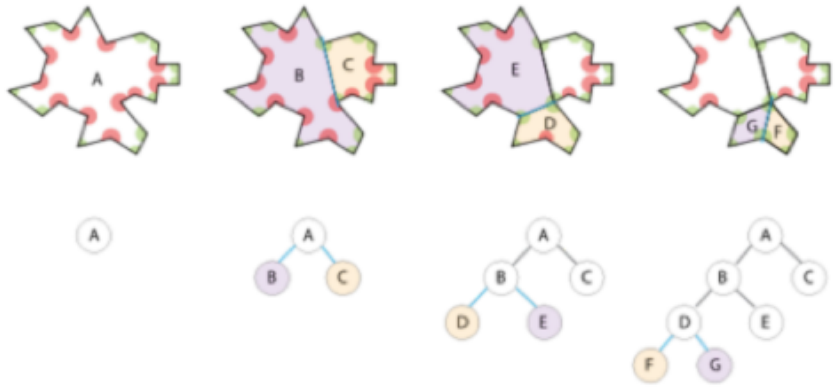


Figure 2.22: Binary space partitioning tree construction illustrated [34].

is also reduced, since the number of possible planes is greatly reduced due to the on-axis splitting plane restriction [6]. Enforcing this condition transforms the general binary space-partitioning tree into a special case called the k-d tree.

K-d Tree

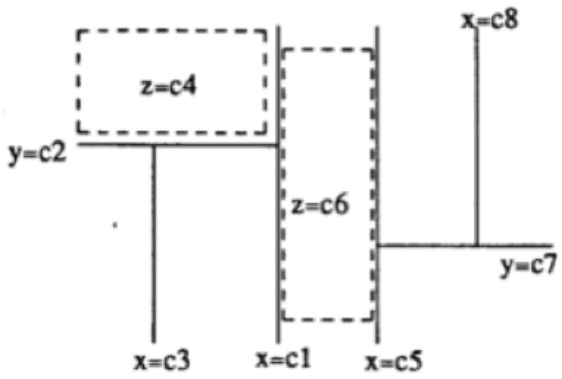


Figure 2.23: The k-d tree is a special case of the binary space partitioning tree in which splitting planes must be on-axis [6].

The goal of k-d tree construction is to partition the object space into two subspaces at each step using planes that are parallel to the x, y, or z axes (Figure 2.23). A plane is chosen for each step based on which of the three available planes will result in the most even division of objects between subspaces. The minimization of object sharing between subspaces is

also a major consideration. All splitting planes are perpendicular to one another, which decreases the computational intensity of ray-plane intersection tests [6].

For computation on a GPU device, the data structures for the BSP tree and k-d tree techniques can both be slightly modified to fit the GPU model used with the bounding volume hierarchy acceleration technique. Since recursion is not available on the GPU, traversal order must be encoded in a vector that is passed as an argument to the GPU. The splitting planes would make up the nodes shown in Figure 2.19 instead of bounding volumes. However, since there tend to be more splitting planes associated with a scene than bounding volumes, more storage space is required to articulate tree composition to the GPU. Performance suffers as a result [31].

2.5 OpenCL

2.5.1 Overview

In recent years, a computing trend has emerged that represents a shift from traditional serial processors towards parallel processors. This movement is driven by the capabilities and limits of the modern semiconductor manufacturing process. While power consumption and heat generation issues restrict traditional processor speed increases, parallel processors can achieve continued performance gains with reasonable power requirements and heat output (Figure 2.24). However, this paradigm shift introduces another issue; in order to take advantage of the additional computing power offered by parallel processors, programs must be written differently. Writing parallel, highly scalable code, which has been historically difficult for a number of reasons, is now absolutely required for high-performance applications.

The diverse and heterogeneous nature of available parallel architectures further complicates the issue. Traditional CPUs have evolved into multicore processors, with two to eight cores per socket. While GPUs were originally created as a single-purpose massively parallel accelerator for graphics applications, they have become increasingly programmable and can now be used for more varied applications. In essence, the GPU has become another

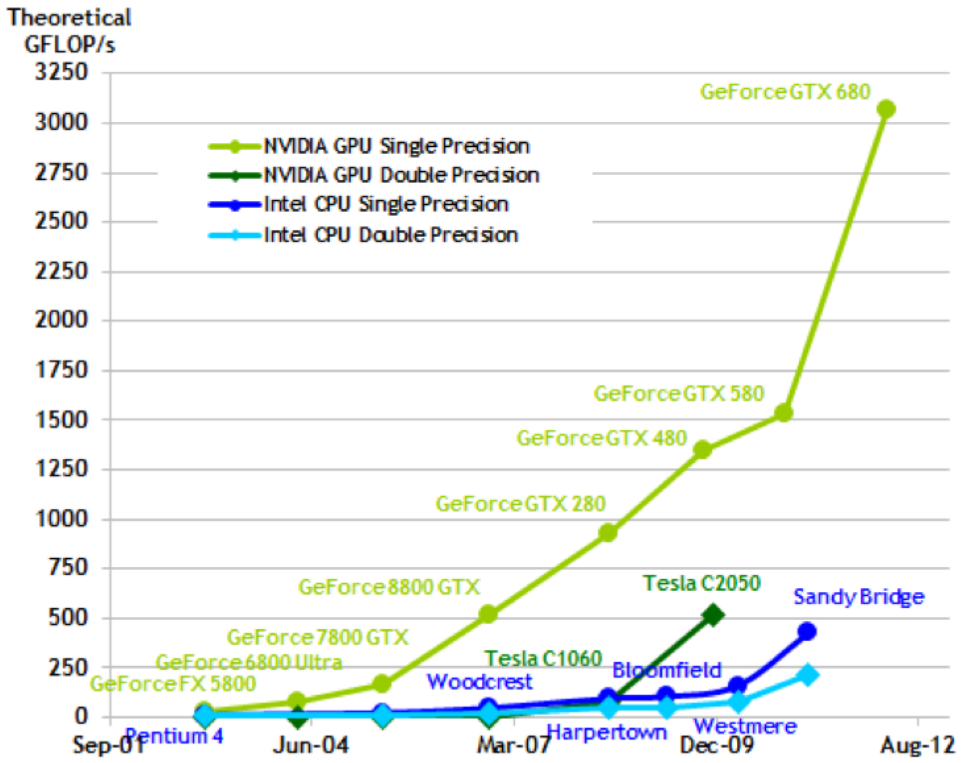


Figure 2.24: GPU performance has rapidly outpaced CPU performance in recent years [19].

flavor of general-purpose processor that is less flexible than the CPU, but contains many more cores per die. Different manufacturers of parallel CPU and GPU architectures do not all subscribe to the same design philosophies, which has lead to an array of incompatible tools and programming models that are required to program each architecture. This in turn leads to a substantial development cost increase for cross-platform projects. Several competing solutions exist to address this set of problems and streamline development for high-performance parallel architectures. These solutions include OpenCL and CUDA; OpenCL has been used for this thesis.

OpenCL allows developers to use a single unified tool chain and language to target all parallel processor architectures currently in use, (i.e. both CPU and GPU platforms are supported). It employs an abstract platform model that conceptualizes all architectures in a similar way and an execution model that supports data and task parallelism across heterogeneous architectures. OpenCL is managed by the Kronos Group, a non-profit organization that also maintains OpenGL. Contributors to the project include

high-profile companies such as AMD, Apple, Intel and NVIDIA, among others. This wide support base ensures that the project remains current with respect to evolving parallel architectures. As with OpenGL, OpenCL provides an API and a runtime system [28].

2.5.2 Platform Model

A hierarchical platform model is used to interface with heterogeneous hardware. As shown in Figure 2.25, the host device coordinates execution by transferring data to and from a set of compute devices (GPU, DSP, or multicore CPU). Compute devices are each composed of an array of compute units, or cores, which are in turn made up of a number of processing elements. Processing elements generally execute instructions as single instruction, multiple data (SIMD) on CPUs and as single program, multiple data (SPMD) on GPUs. The model does not specify what hardware constitutes a compute device, so it is compatible with a variety of diverse hardware types including GPUs, multicore CPUs, and niche processors such as the Cell Broadband Engine [28].

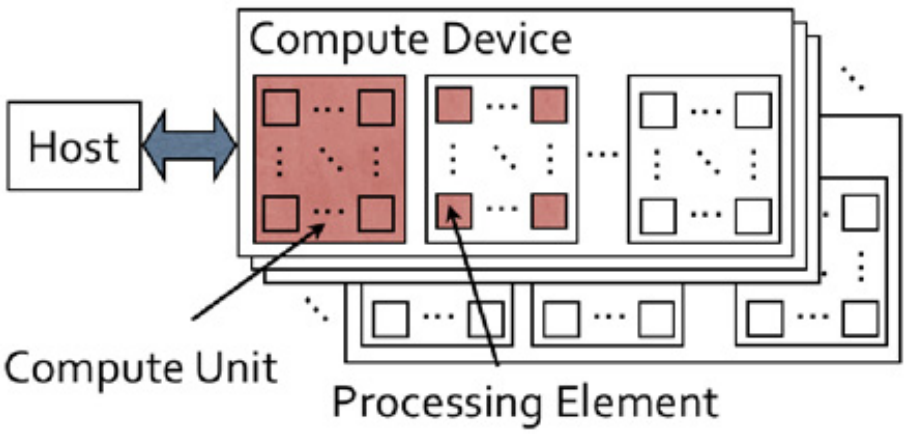


Figure 2.25: OpenCL employs a hierarchical platform model to interface with heterogeneous computing hardware [28].

2.5.3 Execution Model

The OpenCL execution model incorporates both task and data parallelism. Command queues facilitate the movement of data between the host and compute devices, and provide a means to specify dependencies between tasks to ensure that they are executed in the correct order. The OpenCL runtime executes tasks in parallel if all dependencies are satisfied. Tasks are comprised of kernels that apply a single function to a set of data elements in parallel. Synchronization and communication within a kernel are very restricted [28].

The kernel function is applied to a set of independent elements called work items. Work items acquire input data from the host according to an index assigned at queue-time, and each work item executes the same kernel function on its own data. Work items can be grouped together into work groups for local memory sharing and synchronization purposes (Figure 2.26).

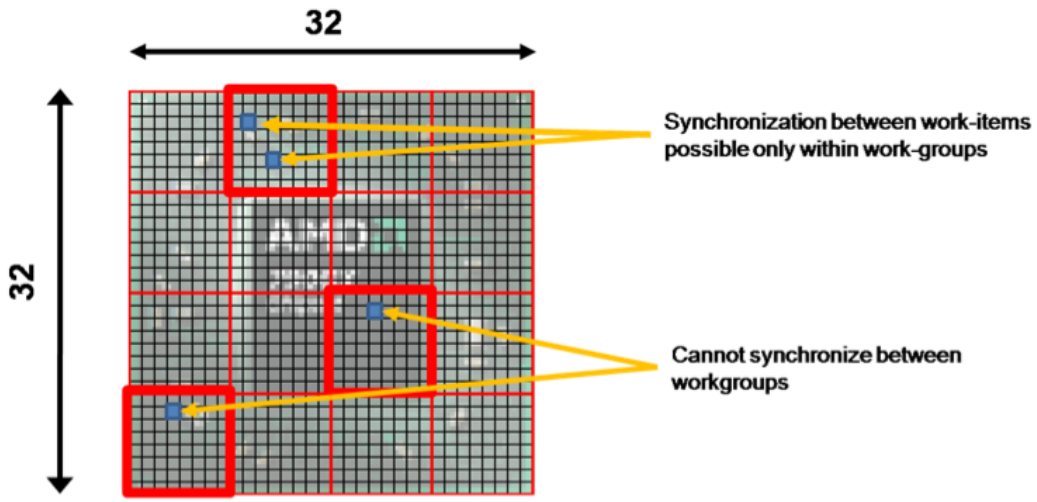


Figure 2.26: The OpenCL kernel is composed of work items that are assembled into work groups. Synchronization and memory sharing is possible only between work items in the same work group [27].

2.5.4 Memory Model

The OpenCL memory model defines four regions of device memory accessible to work-items during kernel execution. Figure 2.27 shows these regions, which include global, constant, local, and private memory. Global memory is accessible to all work items and work groups for both read and write. It is allocated by the host at runtime, and has a large capacity but relatively high memory latency. Constant memory is a subset of global memory that is still accessible to all work items, but only for read operations. Local memory is used for data sharing by work items in a work group, and allows read/write access for all items in the group. Private memory is accessible to only one work-item. Both local and private memory are allocated during kernel execution [27].

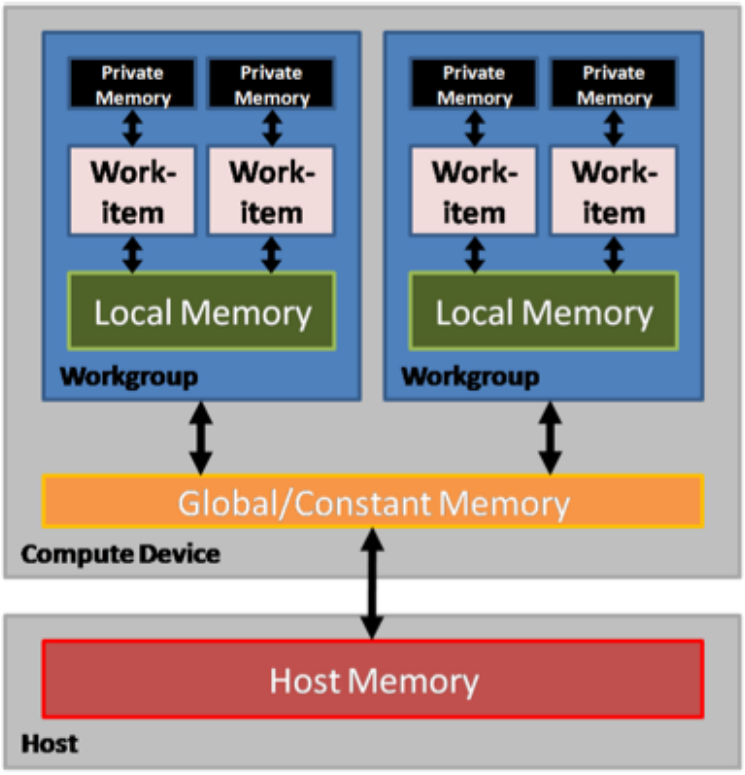


Figure 2.27: The OpenCL memory model defines several regions of compute device and host memory. Memory operations are managed by the host [27].

Host memory and compute device memory are independent of one another, which means that data must be explicitly moved from host memory to global memory to local memory and back. The host controls data movement by enqueuing read and write commands in the command queue [27]. Since memory operations between the host and the compute device are quite expensive, memory management commands should be kept to a minimum for optimum performance.

2.5.5 Executing an OpenCL Program

Figure 2.28 illustrates the process required to execute an OpenCL program. Several OpenCL devices can be leveraged simultaneously through the use of an OpenCL context, which also manages various OpenCL objects including command queues, memory constructs, program objects and kernel objects. In addition, the context is responsible for overseeing kernel execution.

The OpenCL runtime provides a compiler that is used to produce a program executable from OpenCL source code. Each OpenCL program must feature at least one kernel function, which is executed on many independent data members in parallel by the work units within the kernel. Utility functions may also be included and referenced in the kernel function. OpenCL programs are written in a modified version of the C99 standards, which includes several restrictions on recursive functions and function pointers. When moving from a traditional serial implementation to a parallelized OpenCL version, these restrictions can necessitate algorithm modifications to avoid illegal operations.

Since host memory and device memory are not shared, memory objects must be explicitly created in device memory for both input and output data. Input data arrays are then copied to device memory, (single data members may be set as kernel arguments and do not need to be explicitly copied). Once kernel execution is complete and the output data memory objects are populated, output data are explicitly read into host memory.

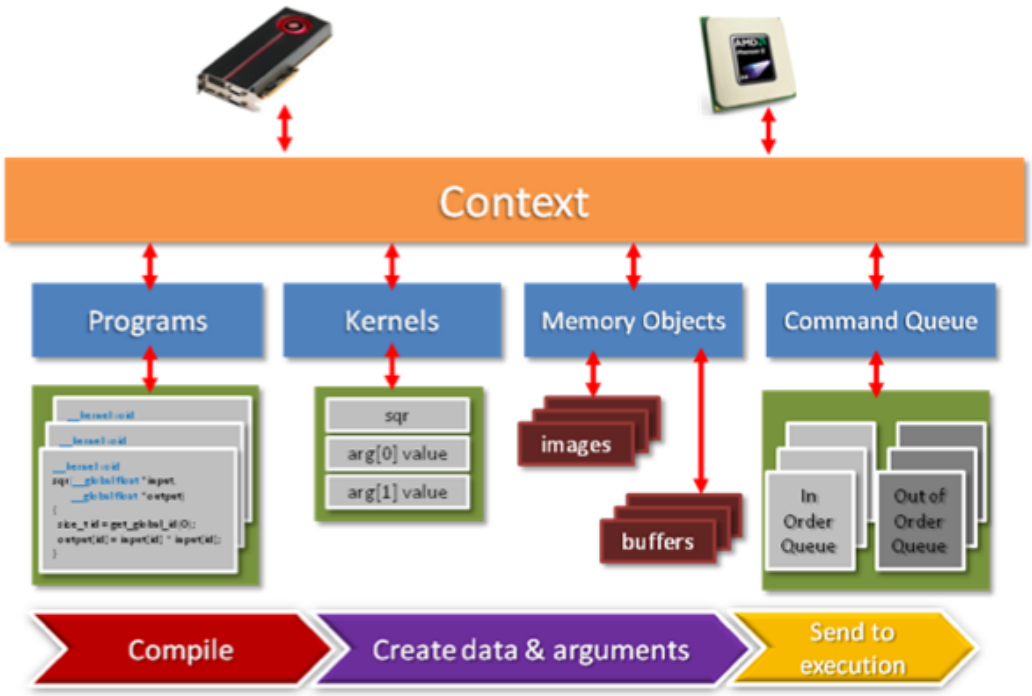


Figure 2.28: OpenCL programs are executed by the host through a series of steps that include compilation, data and argument creation, and transfer to the command queue [27].

The following procedure is used to initialize and execute a standard OpenCL program:

1. Query the host system for OpenCL devices
2. Create a context to associate OpenCL devices
3. Compile programs to run on OpenCL devices
4. Select program kernels to execute
5. Create memory objects in device memory
6. Copy input data to device memory
7. Provide arguments for each kernel
8. Submit kernels to command queue for execution
9. Copy results from device back to host

For programs that require multiple iterations of the same kernel, not all of these commands need to be issued for each execution. For instance, the OpenCL context object and program executable can remain active for the duration of program operation. Memory objects and kernel arguments are persistent, so constant inputs and arguments can be set once at program startup and left as-is for all kernel executions. For graphics programs, even output memory can be allocated once at startup and never be actively read back to the host. This is made possible by the device memory management scheme employed when OpenCL-OpenGL context sharing is used.

2.5.6 OpenCL-OpenGL Interoperability

OpenCL provides several functions that facilitate context sharing, which is the use of OpenGL buffers, textures and render buffer objects as OpenCL memory objects [8]. Context sharing allows the addition of an OpenCL kernel anywhere in the graphics pipeline. Memory buffers can be shared between OpenCL and OpenGL with no data copy operations, and only minor overhead is incurred through context switching. An OpenCL reference object is created for existing OpenGL objects, and ownership is transferred between APIs before use [11]. Only one API at a time is permitted read/write access to the shared memory object.

The OpenGL pixel buffer object (PBO) is a prime candidate for sharing with OpenCL. It allows for fast pixel data transfer between the CPU and GPU using direct memory access (DMA), which does not require CPU supervision [1] (Figure 2.29). Once data are transferred to the GPU via DMA, they remains in device memory and can be modified by both OpenCL and OpenGL without any involvement from the CPU. This feature is very useful for graphics applications that seek to compute pixel data with OpenCL and display them with OpenGL while minimizing overhead.

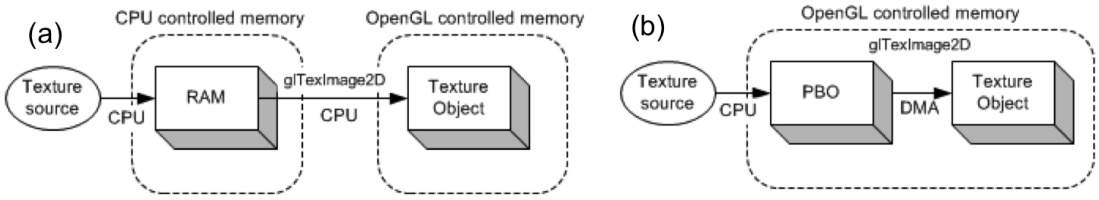


Figure 2.29: Pixel Buffer Object. (a) Memory operation without PBO. (b) Memory operation with PBO [1].

2.6 Previous Work

Computer graphics models that take advantage of the nonuniform acuity of the human visual system for computational speedup or data compression have been proposed and implemented with positive results. While all systems use the same basic perceptual principle for detail reduction, the means of acquiring fixation data and reducing detail vary widely. The most straightforward approach is to use eye-tracking hardware to reduce spatial pixel resolution in regions of low acuity. More subtle methods exist that use a priori knowledge of scene contents and/or user task to reduce resolution in areas on which the user is unlikely to fixate for any length of time. Others still have foregone resolution degradation completely in favor of simplifying polygon meshes outside of the high-acuity region.

2.6.1 Foveal Pyramid

Geisler and Perry's foveal pyramid approach [7] partitions an existing image into distinct regions based on their distance from the current region of interest. The resolution in each of the regions is reduced with a series of low-pass filters, resulting in a multi-resolution image that has full detail in the region of interest and decreases in resolution moving away from this region. Unlike most other techniques, this approach allows for a variable number of resolution levels to account for image content and viewing conditions.

Since the foveal pyramid technique operates on existing images, frame rate is not a primary concern. Instead, the adaptive reduction in resolution decreases the required storage size for each frame in a video and thereby reduces the bandwidth required to transmit the video. While no eye-tracking

hardware was used in Geisler and Perry's implementation, it could be incorporated for more accurate region of interest data. This algorithm achieves a 3x reduction in the amount of data required to represent an image (Figure 2.30).

Figure 2.30: Foveal pyramid image encoding [7]. Resolution in an existing image is spatially reduced according to a model of human visual acuity.

2.6.2 Spatially Adaptive Ray Tracing

Levoy and Whitaker [13] implemented a spatially adaptive ray-tracing system that incorporates real-time fixation data from an eye tracker to produce a multi-resolution rendered image. Their 3D mip-map based algorithm results in a nonuniform sampling distribution across the image plane, with considerably higher ray density in the foveal region.

The viewing region is split into three areas: the foveal region, the peripheral region, and the transition region. The transition region features slightly degraded resolution, and corresponds to the outer region of the user's high acuity from 5° to 9° of visual arc. An eye tracker is used to adjust the sampling distribution according to user fixation. After a five minute preprocessing step, they observed a 4.6x speedup when rendering the 256 x 256 x 109 voxel magnetic resonance scan shown in Figure 2.31.

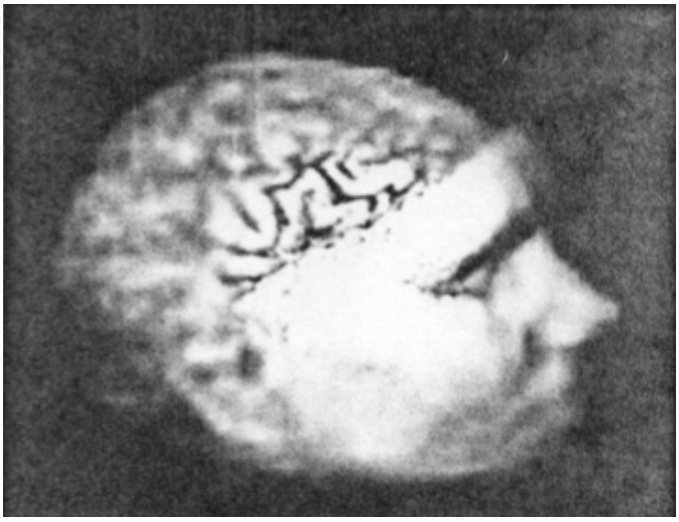


Figure 2.31: Variable resolution ray tracing [13]. Ray density is reduced for areas outside of the fovea. A blurring effect is applied to reduce visible pixilation.

2.6.3 Task-Based Level of Detail Adjustment

Certain features of a scene, such as edges, abrupt changes in color, and sudden movement tend to attract involuntary user attention. Low-level saliency models determine which regions of a scene exhibit these features, and can be used as an alternative to eye-tracking when locating regions of interest. Cater et al. applied a saliency model that includes knowledge of a viewer's visual task in order to render a scene with high resolution in regions of interest and lower resolution elsewhere [4]. Their approach takes advantage of inattentional blindness in addition to nonuniform visual acuity. This phenomenon causes users to fail to notice reduction in image quality in regions not related to the current task, even if those areas fall within the outer region of the fovea. Their approach led to a rendering time of 5.4 hours for the 3072x3072 multi-resolution scene shown in Figure 2.32, compared to a time of 8.6 hours for the same scene in full resolution.

2.6.4 Adaptive Subdivision

Spatial level of detail variation can also be realized through adaptive subdivision of three-dimensional polygon meshes. Reddy developed a system

Figure 2.32: Selective rendering with a task-level saliency model [4]. Resolution is reduced outside of predefined regions of interest that are selected based on user task.

that renders terrain geometry in high detail at the fixation point and a simplified mesh outside of the foveal region [23]. This is accomplished by recursively subdividing the mesh, with regions outside of the fovea terminating earlier than those within. For a terrain model with 1.1 million triangles, the perceptual optimization achieved a 2.7x improvement in rendering time (Figure 2.33). While no eye tracking hardware was used for this model (fixation was bound to the center of the image), Reddy emphasizes the need for such technology to produce an accurate perceptually based system.

A more general-purpose method for adaptive subdivision was proposed and implemented by Murphy and Duchowski [18]. It converts a full-polygon mesh to a variable level of detail mesh through spatial degradation according to visual angle. An eye tracker is used to determine which portion of the mesh to render in full detail while the remainder is rendered using the degraded mesh (Figure 2.34). For a 268,686 polygon Igea mesh, applying this technique allowed for near-interactive frame rates (20 - 30 fps), while

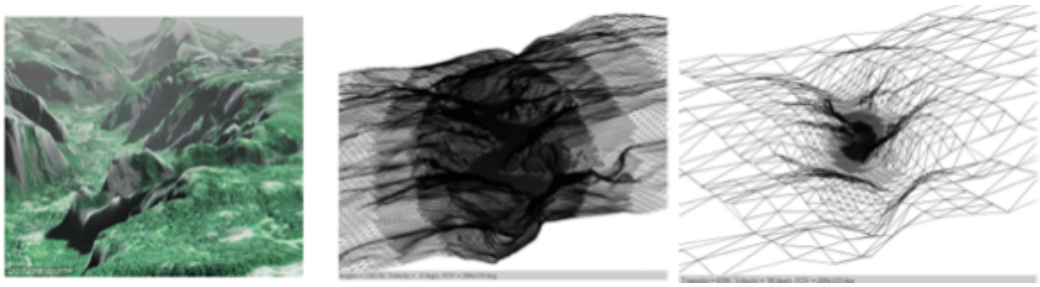


Figure 2.33: Adaptively subdivided terrain [23]. A low-polygon mesh is recursively subdivided to produce higher detail in the foveal region. Subdivision terminates early in the periphery.

frame rate for the full resolution model was ”too low to measure”.

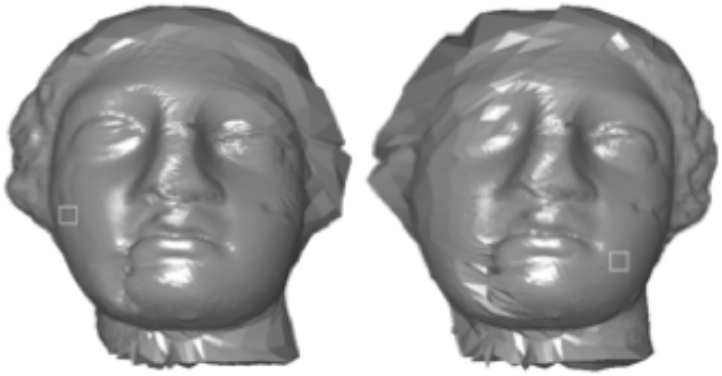


Figure 2.34: Adaptively subdivided model [18]. The full-detail mesh is simplified according to visual angle to generate an alternate low-detail mesh. A combined model is produced on the fly, with the full-detail mesh in the foveal region and the low-detail mesh elsewhere.

2.6.5 Limitations

While many perceptual optimization techniques have shown positive results, existing methods are not well-suited for application in a subtle, perceptually optimized real-time computer graphics architecture. Multi-resolution display models tend to produce noticeable image degradation; according to Levoy and Whitaker [13], “users are generally aware of the variable-resolution structure of the image”. In addition, the nonuniform pixel distribution produced by the multi-resolution approach tends to exhibit poor coherency with regard to the single instruction, multiple data (SIMD) paradigm

employed by modern GPUs. Considering the current trend toward massively parallel computing architectures, this is a major drawback.

Adaptive subdivision comes with a similar drawback; transitioning between the full-detail mesh and the spatially degraded mesh produces motion that is very perceptible to the user's peripheral vision. Task-level saliency models offer excellent computational speedup and low noticeability. However, they are not applicable to the general case, where the user task may be complex and regions of interest are not guaranteed to be consistent or easily identifiable. Furthermore, automatic prediction of attention regions has been shown to be unreliable [15].

The perceptually optimized rendering framework presented in this thesis also leverages the difference in acuity between the foveal and peripheral regions of the field of view to provide computational speedup while more effectively preserving perceived image quality compared to spatial degradation techniques. Chapter 3 describes the design of the perceptually optimized framework, including implementation details for the novel RRM technique.

Chapter 3

System Design

3.1 Overview

The Perceptually Optimized Real-Time Computer Graphics framework was developed to take advantage of the nonuniform acuity of the human visual system for computational speedup in different graphics applications. Over the course of development, the novel Refresh Rate Modulation technique emerged as an effective means of achieving real-time frame rates with Whitted's classic ray-tracing algorithm, and it became the primary area of interest. The main focus of this thesis is therefore the implementation of a high-performance GPGPU ray tracing engine that incorporates the novel RRM technique as well as more traditional acceleration techniques. A secondary investigation is conducted regarding the performance benefits of adjusting physics engine error tolerance for collisions in the user's periphery.

3.2 Ray-Tracing Engine

3.2.1 Overview

Ray-tracing is a well-established method for rendering three-dimensional scenes [33]. The algorithm models the approximate path of light in reverse, flowing from the camera to objects in the scene. When a light ray intersects an object, the associated pixel is filled with the color of the object at that point. For reflective and refractive objects, additional rays are spawned recursively at the point of intersection.

The ray-tracing algorithm is very computationally intensive, which has

historically prevented it from being used for real-time applications. Approximately 75% of the time required to render simple scenes is allocated to computing ray-object intersections, with this number increasing for scenes with a large number of objects. A performance speedup can be realized by reducing the number of intersection tests per ray or the overall number of rays computed. This system is built on a basic ray-tracing framework, and is designed to reduce the number of rays that need to be computed by taking advantage of the differences in visual acuity between the foveal and peripheral vision. It also includes a number of traditional optimizations that reduce the number of intersections per ray as well as the time required to compute each intersection.

3.2.2 Structural Acceleration

The bounding volume hierarchy (BVH) is one effective method of organizing scene object data to reduce ray-object intersection calculations per ray. Each polygon in a mesh is encapsulated within a bounding volume; the framework uses a sphere, which has a relatively low intersection cost. This set of volumes is paired and encapsulated within larger volumes until only one volume remains. This volume now contains a hierarchy that represents all geometry in the mesh. Ray intersections on the entire mesh are performed using the hierarchy. If a ray intersects the top level bounding sphere, its children are recursively checked for intersection. The BVH scheme eliminates all but one sphere intersection test for the majority of rays that do not actually intersect the mesh. See Section 2.4.1 for more details.

In order to send the bounding volume hierarchy to an OpenCL kernel, it must be flattened into a one-dimensional array. The flattening algorithm described by Thrane [31] serves as a basis for this method. The bounding volume hierarchy is traversed depth-first, and a traversal index is assigned to each node to indicate traversal order. Each node is also given an escape index, which indicates where to go next if its child nodes are to be ignored. Leaf nodes require a third index that corresponds to the mesh triangle that they encapsulate. Once these values are assigned, all bounding volumes can be placed in a linear array in the order of their traversal indices. The

flattened BVH and triangle data are written to a file and loaded directly on subsequent executions to accelerate program startup.

Traversal of the hierarchy begins by intersecting the current ray with the top level bounding volume. If an intersection occurs, the counter used to index into the BVH array is incremented, which moves to the next bounding volume in the traversal. If there is no intersection, the counter is set to the node's escape index to skip its children and move to its nearest sibling node. Traversal terminates either when the triangle within a leaf node is intersected, or when the rightmost bounding volume at any level is found to have no intersection with the current ray.

3.2.3 GPU Acceleration

General purpose GPU acceleration has emerged as a compelling means of reducing intersection computation time for high performance ray-tracing systems. This framework places all ray-tracing logic, including ray generation, intersection tests, illumination, texturing and shading, in a single OpenCL kernel for execution on the GPU. A read-only input array of work unit structures holds coordinate data that tells each OpenCL work item into which onscreen pixel to write its result. Perceptual optimization leads to two separate pixel groups, which prevents the 1:1 correspondence between work items and onscreen pixels that would normally allow natural OpenCL kernel indexing to handle coordinates. Pixel work units are allocated once at the begin of operation, and are arranged in horizontal 4×1 strips for increased coherency. A writable OpenGL Pixel Buffer Object (PBO) is shared with the OpenCL kernel and stores pixel data from each frame for display onscreen. The PBO remains in device memory throughout program execution, which bypasses costly GPU-CPU communication.

While faster constant memory is used for most input data, the small size of the local memory region where constant memory resides necessitates the use of global memory for potentially large buffers such as triangle and bounding volume hierarchy arrays. Branching has been avoided when possible due to the relatively high cost associated with GPU branch misprediction. For example, some rarely-used benchmarking features are toggled

with hardcoded Boolean values in the kernel. While this implementation does sacrifice flexibility, the compiler is able to remove unused branches before runtime for improved performance.

3.2.4 Secondary Rays

A secondary ray stack is used to avoid the recursive function calls featured in the classic serial ray tracing algorithm. Each OpenCL work item has access to a private stack for secondary ray processing, the size of which depends on how many reflection and transmission rays need to be processed. While a large stack can be used to ensure sufficient space for most scenes, stack size is easily adjustable and should be tailored to match the current scene for optimum performance.

Private read and write pointers are maintained within each work item. The work item enters the stack after primary and secondary ray calculations are complete if secondary rays were added to the stack and reflection is enabled. When a secondary ray is spawned, it is stored in the stack and the write pointer is incremented. Each ray in the stack is processed in order and, if the ray intersects a reflective and/or transmissive object, new secondary rays are added to the stack. This continues either until the read pointer is equal to the write pointer, or until the capacity of the stack is exhausted.

While a stack structure is used to handle secondary rays in the final implementation, an alternative multi-pass strategy was explored in detail. It involves processing secondary rays with consecutive kernel calls (Figure 3.1), which allows OpenCL to reconfigure work distribution for each reflection or refraction step in order to maintain GPU saturation. This contrasts with the more traditional stack-based approach, where processing elements that do not generate secondary rays sit idle until those with secondary rays finish computation. Secondary ray output for most scenes is very sparse, so the CPU-side host program compresses the output array before initiating the next kernel pass. While this results in faster kernel execution, the GPU-CPU communication required to exchange and compress secondary ray data is costly and leads to decreased frame rate. However, it does cause the RRM technique to yield a better speedup (6.3 with multi-pass secondary rays vs

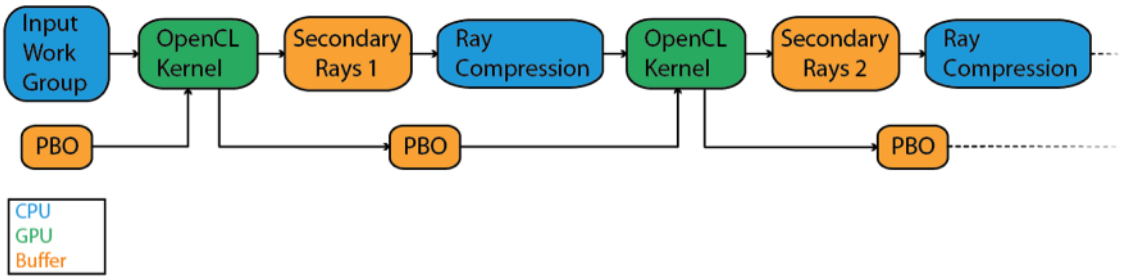


Figure 3.1: Multi-pass secondary ray processing. Instead of using recursion or a secondary ray stack, multiple passes are made with the same OpenCL kernel.

2.3 with the stack-based approach). As such, further investigation is warranted.

3.3 Perceptual Optimization

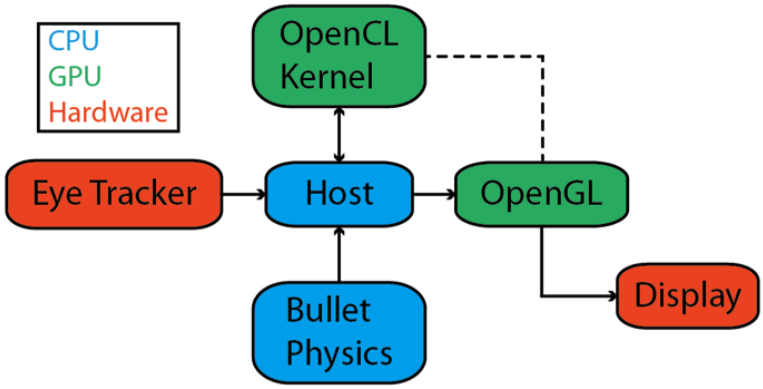


Figure 3.2: Runtime data flow. The framework is composed of several subsystems that work in tandem to produce perceptually optimized computer graphics.

Figure 3.2 shows an overview of the perceptually optimized framework. The host serves as a central hub and handles initialization and interprocess communication. It also maintains object and camera positional data, and manages the Refresh Rate Modulation cycle. The OpenCL kernel contains the full rendering algorithm, which writes to a persistent pixel buffer that is shared between OpenCL and OpenGL in GPU memory. OpenGL is responsible for writing this buffer to the display each frame and initiating the

rendering process for the next frame. Eye tracking hardware provides real-time fixation data in the form of X and Y screen coordinates, which are passed from the host to the OpenGL kernel to update foveal position. If physics functionality is enabled, the Bullet Physics engine drives positional data for all scene objects.

The eye-tracker used in this project is a Mirametrix S1 eye-tracking device operating at 60 Hz with gaze position accuracy of less than 1°. While the data it provides are reasonably accurate, like all eye-trackers it does exhibit some degree of noise. Using raw fixation data detracts from the user experience, because peripheral vision is extremely sensitive to motion [16]. To rectify this issue, an auxiliary smoothing filter has been placed between the eye tracker and the host.

The standalone smoothing filter was developed by Sean Xu as part of his Eye Tracking Framework thesis project. The eye-tracking hardware broadcasts fixation data on a TCP socket in real-time, which is received by the filter and incorporated into a running average that is rebroadcast on a different TCP socket. A thread spawned by the host connects to this socket and listens for updated fixation values. At the beginning of each frame, the current fixation value is provided to the OpenCL kernel as an argument so the onscreen foveal position can be adjusted.

3.4 Refresh Rate Modulation

The primary contribution of this work is the Refresh Rate Modulation technique for perceptually adaptive level of detail adjustment. The display is split into 2 segments: the fovea and the periphery. The foveal region is a small inset pixel group that corresponds to the high acuity region of the human visual system, and is updated once every frame. The work units in the foveal region are densely packed, with one work unit for each onscreen pixel in the region (Figure 3.3). The peripheral region takes up the remainder of the image, and lies within the lower-detail portion of the user's field of view. This region is fully refreshed only once every N frames, which results in a substantial computational speedup. Work units in this group are sparsely distributed, with one work unit for every N onscreen pixels in the region.

Work units in each region are arranged in groups of four consecutive pixels (pixel strips) to maximize SIMD coherency and improve performance.

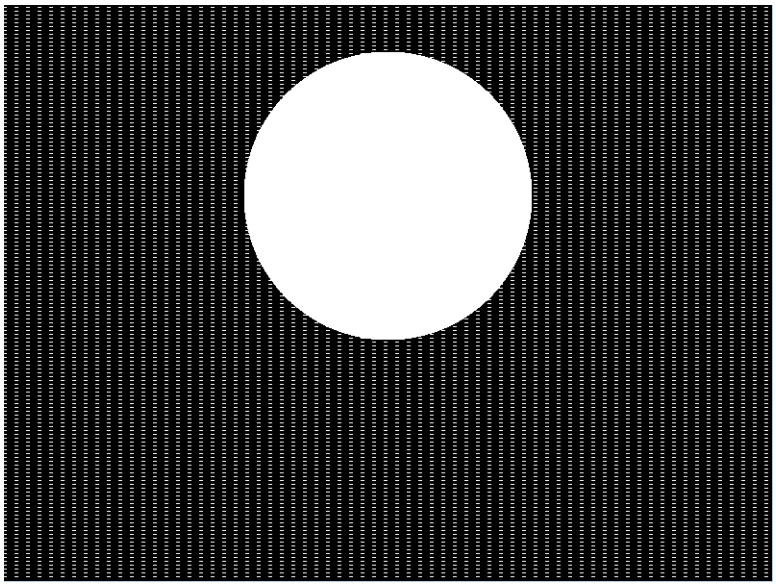


Figure 3.3: Work group layout. The display area is segmented into a dense inner region (the fovea) and a sparse outer region (the periphery). White pixels represent work units computed during a single frame

Figure 3.4 illustrates the Refresh Rate Modulation technique. A single pixel strip is processed at each frame, while the rest of the pixel strips in the refresh group maintain data from previous frames. When the red marker is reached, a full cycle is complete, and rendering for the next frame begins again at the green marker. Pixel strips in the foveal region undergo a full cycle each frame, so they are updated in real-time. Units in the peripheral region undergo a full cycle only once every N frames.

Applying the Refresh Rate Modulation technique leads to an effect in which the portion of the display that is viewed by the fovea is rendered in crisp detail, while the rest of the display is subtly fragmented. This fragmentation occurs only when the camera or scene elements are in motion; if scene movement ceases, the display naturally resolves a full-detail image after N frames with no additional handling or overhead. This results in a full-resolution rendering after less than half a second for applications with a real-time frame rate.

The RRM technique avoids the post-processing step that is required by

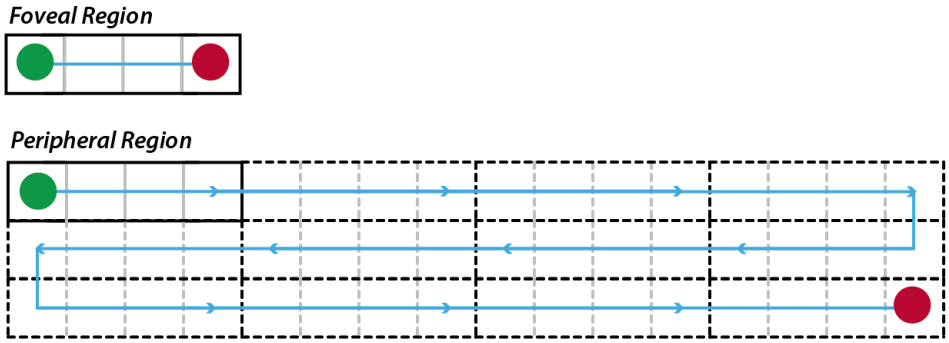


Figure 3.4: Refresh rate modulation. Pixels in the foveal region are updated every frame for real-time rendering, while those in the peripheral region are updated less often for computational speedup. Refresh groups are composed of N pixel strips, where a pixel strip is defined as four adjacent pixels. One pixel strip in each work group is refreshed per frame in the peripheral region, where the individual pixel strip to be updated cycles between the N strips in the work group (which is why it takes N frames to resolve a full-resolution image when there is no scene motion). In this image, pixels are surrounded by a gray box, while sets of four pixels are grouped into pixel strips with a black box. Pixel strips that hold data from previous frame are outlined with dotted lines, while the single pixel strip that is being updated in the current frame is outlined with solid lines. N=12 for the refresh group shown here.

traditional spatial degradation techniques to reduce visible pixilation, and thereby prevents a great deal of processing overhead. Refresh order within the refresh groups can be adjusted on the fly to adjust fragmentation style for scene contents and movement, (e.g. horizontal, serpentine, scattered), and work group size may be decreased in real-time to reduce fragmentation in high-motion scenes or increased to maximize performance. Movement of the foveal region is accomplished within the GPU kernel by simply adding the current fixation position to each work unit position. This allows the entire input work group GPU memory buffer to remain unchanged throughout execution, and avoids costly CPU to GPU memory operations. The peripheral region is also shifted in the same manner each frame. A refresh cycle offset is added to the coordinates of all work units in the peripheral region to update the appropriate pixels within the refresh group.

A persistent OpenGL pixel buffer object is maintained and shared between OpenGL and OpenCL for efficient memory management. Pixel buffer objects allow pixel data to be stored in high-performance graphics memory

on the GPU, and enable fast data transfer to and from the graphics card through direct memory access (DMA) without CPU involvement [1]. Since the buffer is not flashed at the beginning of each frame, peripheral data from previous frames can be leveraged to provide meaningful context for the real-time contents of the foveal region.

The OpenCL kernel takes advantage of the lack of data dependency between pixels in the ray-tracing algorithm to perform calculations for all pixels in parallel. Various data are passed to the kernel from the host for use with the ray-tracing algorithm, including fixation and camera position, frame dimensions, benchmarking parameters, and scene geometry. OpenCL automatically determines local work group size and distributes the workload among kernel work groups. Each work item in the kernel undergoes the following process each frame:

1. Retrieve X and Y pixel coordinates from input work unit array.
2. Shift coordinates by fixation position (foveal group) or refresh cycle offset (peripheral group).
3. Calculate ray originating at eye point and passing through shifted coordinates.
4. Intersect ray with all spheres, all planes, and subset of triangles using bounding volume hierarchy - if no intersections are found, proceed to Step 7. Secondary ray data at point of intersection is calculated here.
5. Spawn shadow ray to light source.
6. Use intersection point, surface parameters from Step 4 and shadow data from Step 5 as input to Phong shading model.
7. Fill pixel with object color from Step 6 or background color from Step 4.
8. Store secondary rays in stack.
9. If the stack read pointer has not yet reached the write pointer and stack space is not exhausted, loop to Step 4 for next stack entry; otherwise terminate.

Once all work items have finished computation, the OpenCL kernel exits and returns control to the host. The host passes ownership of the shared PBO from OpenCL to OpenGL, and instructs OpenGL to write the contents to screen.

3.5 Issues Associated with Spatial Degradation

A more conventional multi-resolution ray-tracing framework was developed near the beginning of this project, and motivated development of the RRM technique. Analyzing performance differences between execution on the CPU and GPU yielded valuable insight regarding GPU characteristics as well as bottlenecks in the algorithm itself. Figure 3.5 shows a sample image from this original framework with a grid feature enabled to highlight the various resolution levels.

While results for this implementation were quite promising on the CPU and on an older commodity GPU, performance was less than impressive on a newer high-performance GPU. There were two primary reasons for this: complex CPU-side work group management between frames, and irregular work groups that are not well-suited to SIMD. In addition, this technique leads to pixilation even when there is no movement in the scene, which in turn requires a post-processing blur effect to remove visible seams. While post-processing effects can be achieved easily with the OpenGL Shading Language (GLSL), frequent switching between OpenCL and GLSL contexts on the GPU is very costly and results in a massive performance drop. Observing these characteristics led to a focus on minimizing work group management, simplifying the work group layout and achieving acceptable perceived image quality without post-processing.

3.6 Perceptually Optimized Collision Detection

In order to demonstrate the flexibility of the framework, a perceptual collision detection subsystem has been developed that renders realistically moving geometry using the ray-tracing engine and the Bullet Physics Library.

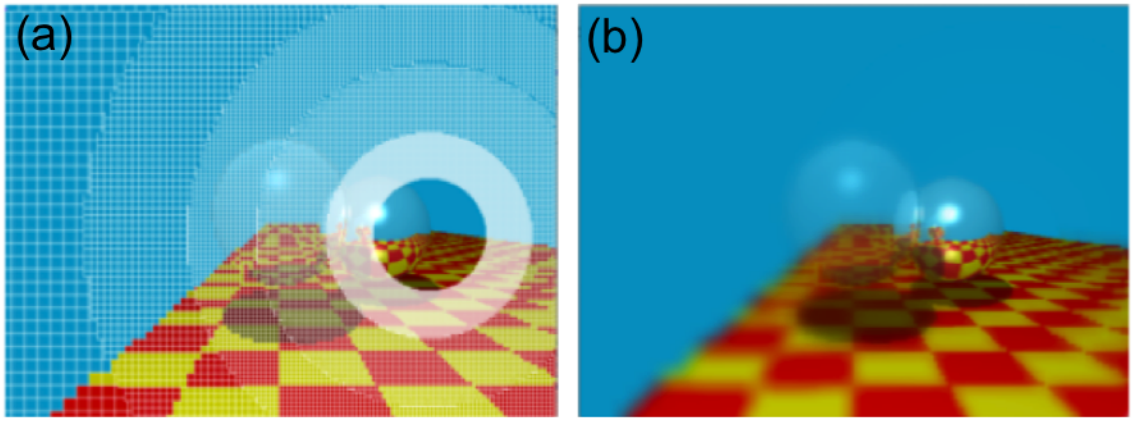


Figure 3.5: Spatial degradation results in less optimal performance and requires a costly blurring effect that is still noticeable to peripheral vision. (a) Whitted scene with grid to reveal multi-resolution layout (b) Whitted scene with blur. Static scene is shown for both images.

O'Sullivan [20] showed that interruptible collision detection can significantly reduce the time required for physics calculations while maintaining plausible scene motion. The system takes advantage of the bounding volume hierarchy based collision detection system that Bullet Physics provides for use with static polygonal meshes. A performance improvement is gained by using only the top level of the BVH for collisions, which is subtle enough to be imperceptible to the periphery.

The Bullet Physics Library (BPL) automatically handles collisions between a variety of object types, including the triangle meshes, planes, and spheres used in this framework [25]. On program startup, all scene object types, sizes and locations are registered with the BPL. Parameters such as object weight and world gravity are also provided at this time. At the beginning of each frame, the host (Figure 3.2) instructs the BPL to move forward one step in time. After object movement and collisions are processed by the BPL, the host retrieves updated object locations and passes them to the OpenCL kernel via a buffer in device memory.

The scene used to gather speedup data contains one high-polygon mesh, one visible ground plane and a variable number of spheres. The BPL applies a bounding volume hierarchy to the high-polygon mesh for accelerated collision detection. Due to the prohibitively high cost of calculating

new bounding volume hierarchy data at each frame for a moving mesh, the BPL requires that BVH meshes remain static. All spheres are subject to BPL gravity. If no spheres collide with the mesh for several consecutive frames, a negligible speedup will be realized as only the top level of the bounding volume hierarchy is queried. To avoid this, invisible collision planes are placed around the mesh and spheres.

The next chapter examines the performance and noticeability of the novel RRM technique as part of the perceptually optimized framework, and shows a number of sample images generated using the framework. It also provides a performance benchmark for the perceptually optimized collision detection system.

Chapter 4

Results

4.1 Overview

The quality of a perceptual optimization technique must be assessed from two perspectives: computational speedup, and perceptual subtlety. The RRM approach requires a specialized metric for computational performance, since the onscreen position of the dense foveal region can have a pronounced effect on frame rate for scenes with nonuniform complexity. To account for this, one frame is rendered for each possible foveal position, and the results are averaged to produce a representative overall frame rate.

A number of benchmarks have been conducted to measure the performance impact of different aspects of the perceptual framework. All tests were performed on a system with a 3.6 GHz Intel Core i7 processor and a Radeon HD 7970 GPU, except where noted otherwise.

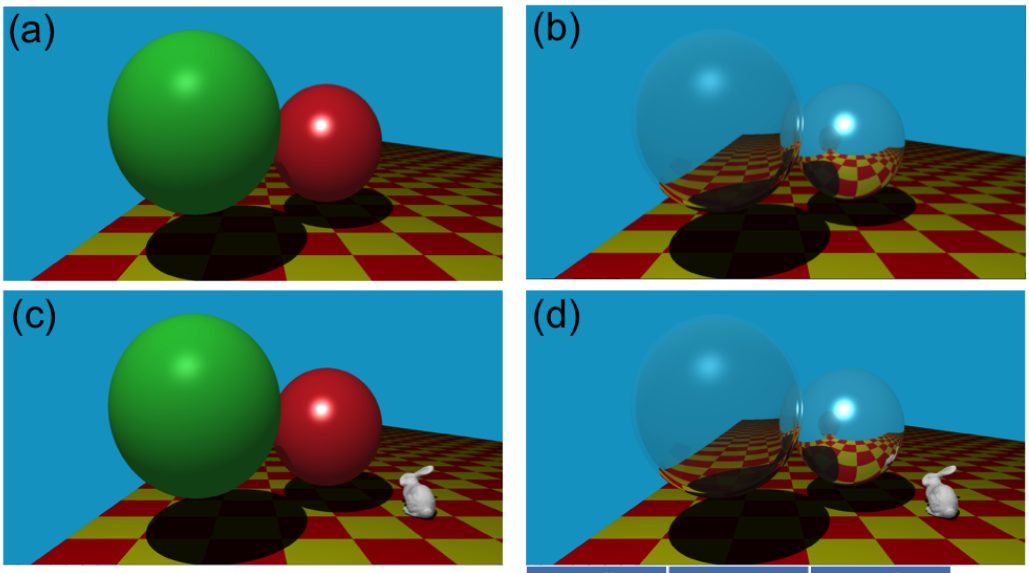
The perceptual subtlety of RRM was measured with a small pilot study. Subjects used the framework in full resolution mode and with RRM enabled, and rated the noticeability of RRM on rendered image periphery. The study also examined the effect of refresh group size on noticeability.

The quality of the perceptual physics optimization was gauged by first measuring the time required for physics calculations. The optimization was then enabled, and calculation time was measured again.

4.2 Benchmarks

4.2.1 Refresh Rate Modulation

In order to assess the overall effectiveness of Refresh Rate Modulation, frame rate measurements were taken for four scenes with and without RRM, with all acceleration techniques enabled. A 1920×1080 frame size was used, with a foveal radius of 270 pixels and 3×4 (or N=12) refresh groups. These settings displayed the greatest balance between performance and perceptibility in preliminary tests.



(f)

	Full Resolution	RRM	Speedup
Whitted Scene (no secondary rays)	144 fps	279 fps	*1.94*
Whitted Scene (with secondary rays)	111 fps	252 fps	*2.27*
High Poly Scene (no secondary rays)	41 fps	140 fps	*3.41*
High Poly Scene (with secondary rays)	52 fps	100 fps	*1.92*

Figure 4.1: Performance results for selected scenes rendered at 1080p resolution. (a) Whitted scene without secondary rays. (b) Whitted scene with secondary rays. (c) High polygon scene without secondary rays. (d) High polygon scene with secondary rays. (f) Performance results for each scene at full resolution and with RRM enabled. A 3×4 refresh group is used for RRM.

Figure 4.1 shows the computational speedup that RRM achieves for the

classic Whitted scene as well as a high polygon scene that features a 5,000-polygon model of the Stanford Bunny, each with and without secondary ray effects. The high polygon scene without secondary rays offers the best speedup. The framework handles the high polygon scene very well with RRM enabled, achieving a frame rate that is considerably better than real-time both with and without secondary rays.

4.2.2 Refresh Group Size

Both performance and perceptibility tests were conducted for a variable refresh group size, since this parameter has a less intuitive impact on noticeability than other factors such as fovea size. Figure 4.2 shows the performance impact when N is increased from the default of 12 to a maximum of 132 for the high polygon scene without secondary rays (Figure 4.1c). Frame rate shows a general upward trend, with a maximum increase of 50 fps. See Section 4.3.2 for perceptibility results.

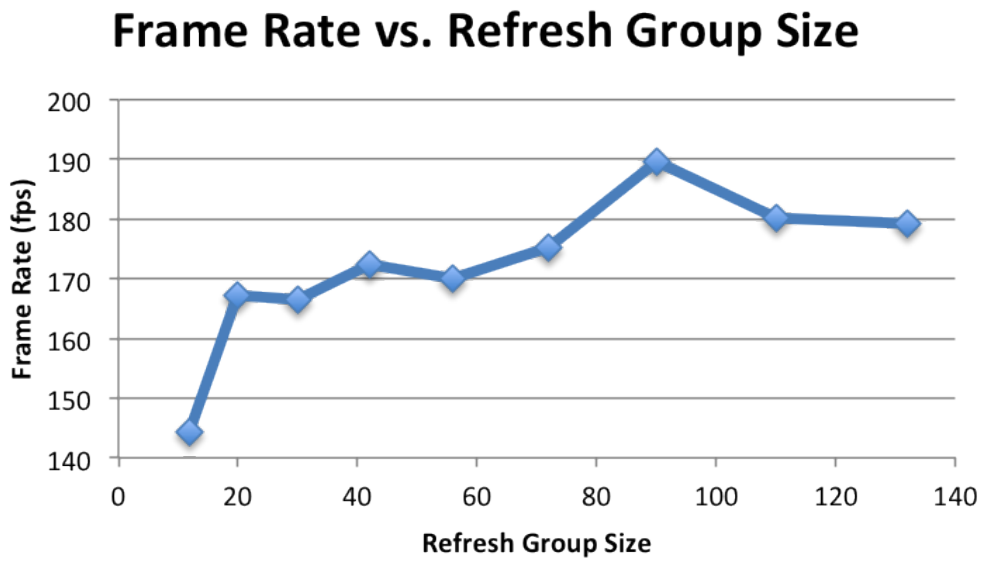


Figure 4.2: Performance of RRM for different refresh group sizes. Increasing group size reduces the number of pixels that must be updated each frame and improves performance.

4.2.3 Frame Resolution

As with any graphics application, the resolution of the rendering window has a large effect on frame rate since it alters the amount of information that must be calculated for each frame. Several tests were performed with common resolutions to quantify the magnitude of this effect on the framework both with and without RRM enabled. Figure 4.3 illustrates the relative size of each resolution. As shown by the results in Figure 4.4, RRM is more effective for larger resolutions, with a maximum speedup of 3.62 for 1024×768 pixels. Raw frame rate peaks at 244 fps for an 800×600 frame with RRM enabled. RRM has essentially no effect for the 640×480 frame size. This is likely due to lack of saturation on the GPU.

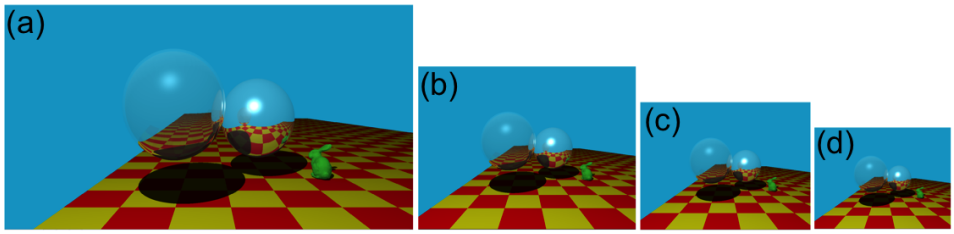


Figure 4.3: Relative frame sizes for common resolutions. (a) 1920×1080. (b) 1024×768. (c) 800×600. (d) 640×480.

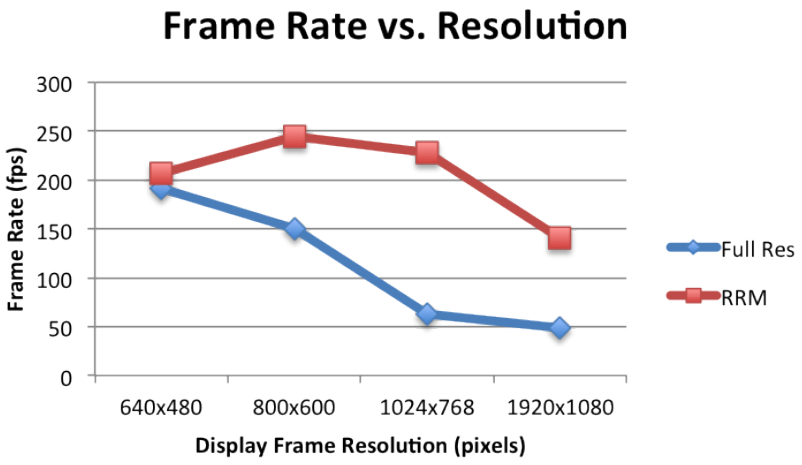


Figure 4.4: Performance for common resolutions with and without RRM enabled.

4.2.4 Foveal Radius

Changing the size of the fovea also alters the volume of information that must be computed for each frame. A number of different foveal radii have been tested to establish the associated performance trend (Figure 4.5). All other RRM benchmarks use a foveal radius equal to $\frac{1}{4}$ of the frame height, (e.g., 270 pixels for a 1080p frame).

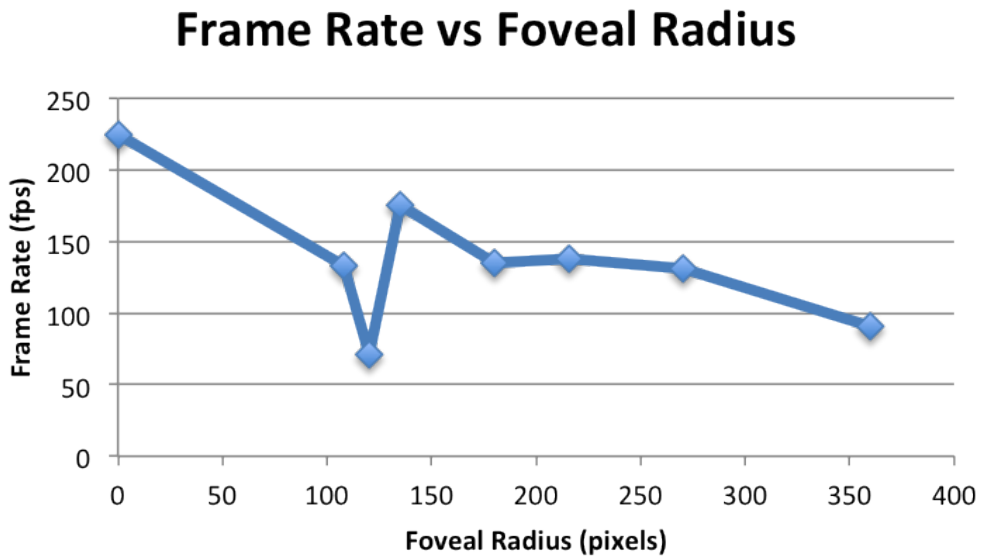


Figure 4.5: Performance of RRM for different foveal radii.

The performance gain from reducing the foveal radius comes with an obvious perceptual cost. However, for large-format monitors that have a greater pixel size, a smaller foveal radius can produce a viable perceived result while maintaining a higher frame rate. An unusual performance dip occurs from a foveal radius of 108 to 120, followed by a spike from 120 to 135. The order construction for the OpenCL pixel location buffer is responsible. Since the fovea and periphery are built separately, GPU work items that process pixels that are on the fovea-periphery border tend to be in different work groups, which can result in poor SIMD coherency. Different foveal radii result in different border layouts, and the system is more sensitive to this when the radius is small.

4.2.5 Bounding Volume Hierarchy

The bounding volume hierarchy greatly reduces the average number of intersection calculations per ray. Figure 4.6 shows the time required to render a single frame of the full resolution high polygon scene with reflection and transmission, both with and without BVH enabled.

When the BVH is used, the scene is rendered in real-time. When it is not used, rendering time increases to 10 seconds per frame. This illustrates that traditional structural acceleration is vital to achieving real-time results, even for a high performance GPGPU ray-tracing engine.

BVH Disabled	BVH Enabled	Speedup
10.83s	.0242s	*447.52*

Figure 4.6: Impact of BVH technique on single frame render time for full-resolution high polygon scene with secondary rays.

4.2.6 Polygon Count

Figure 4.7 shows the performance of the framework for several different versions of the Stanford Dragon, which range from 5,000 triangles to 50,000 triangles at 640×480 resolution. Each model is shown in Figure 4.8 with and without reflection and transmission enabled (frame rates for each are labeled). Secondary ray effects are cut short for the 50,000 triangle model due to the relatively small buffer capacity of local memory on current GPUs. Maximum secondary ray stack size is reduced from 20 to 5 for the 50,000 triangle model to avoid this issue.

Model complexity was increased past the point at which the framework can produce real-time results in order to provide an adequate means of comparison with previous work. The effectiveness of RRM was also measured for each model; while RRM is only perceptually viable for real-time frame rates, the speedup should still apply for more capable future GPUs.

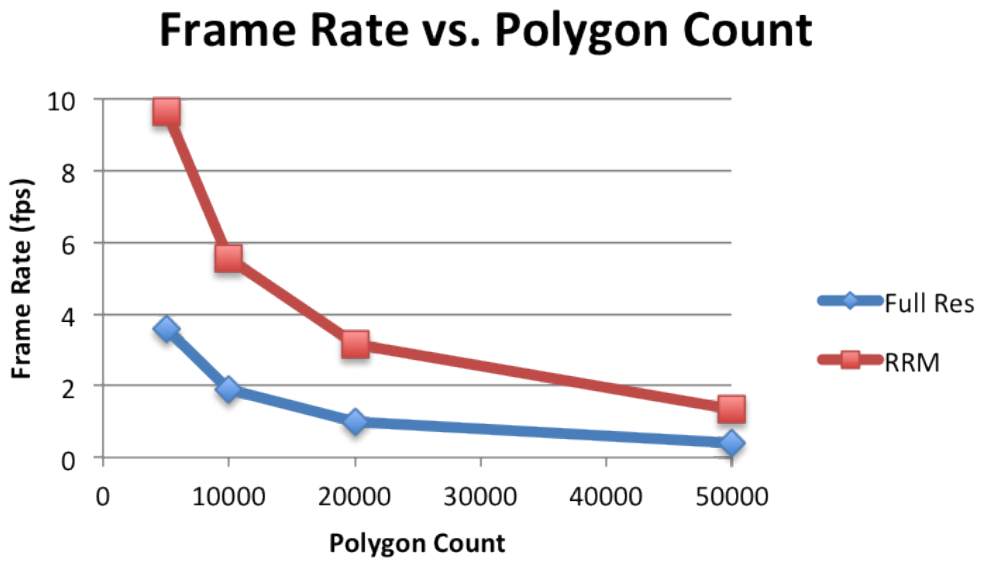


Figure 4.7: Performance for Stanford Dragon model with different polygon counts.

4.2.7 Object Size

The onscreen size of a polygonal mesh has a large impact on performance. A small onscreen size can result in excellent performance even for extremely high polygon meshes, as the mesh is treated as a single sphere for a large majority of onscreen pixels due to the BVH. However, as object size increases a larger percentage of pixels must delve deeper into the BVH to test for intersection with the object, which in turn causes a significant performance drop. The results shown in Figure 4.9 illustrate this effect for a 5,000 polygon model of the Stanford Bunny rendered at 640×480 pixels.

The model is slowly enlarged, starting at 1% of total onscreen pixels and proceeding to 100% (Figure 4.10). An extremely large drop in frame rate occurs from 1% to 10% of pixels. While this intuitively makes sense as it represents 10x increase in object size compared to at most a 2x increase for other steps, it could have a profound perceptual impact. A mesh that occupies 1% of onscreen pixels is not unreasonable for a larger frame – the Stanford Bunny in the high polygon scene used for several earlier benchmarks occupies only 0.68% of the 1920×1080 frame. The scene runs smoothly at the default camera distance, but zooming in the mesh results in

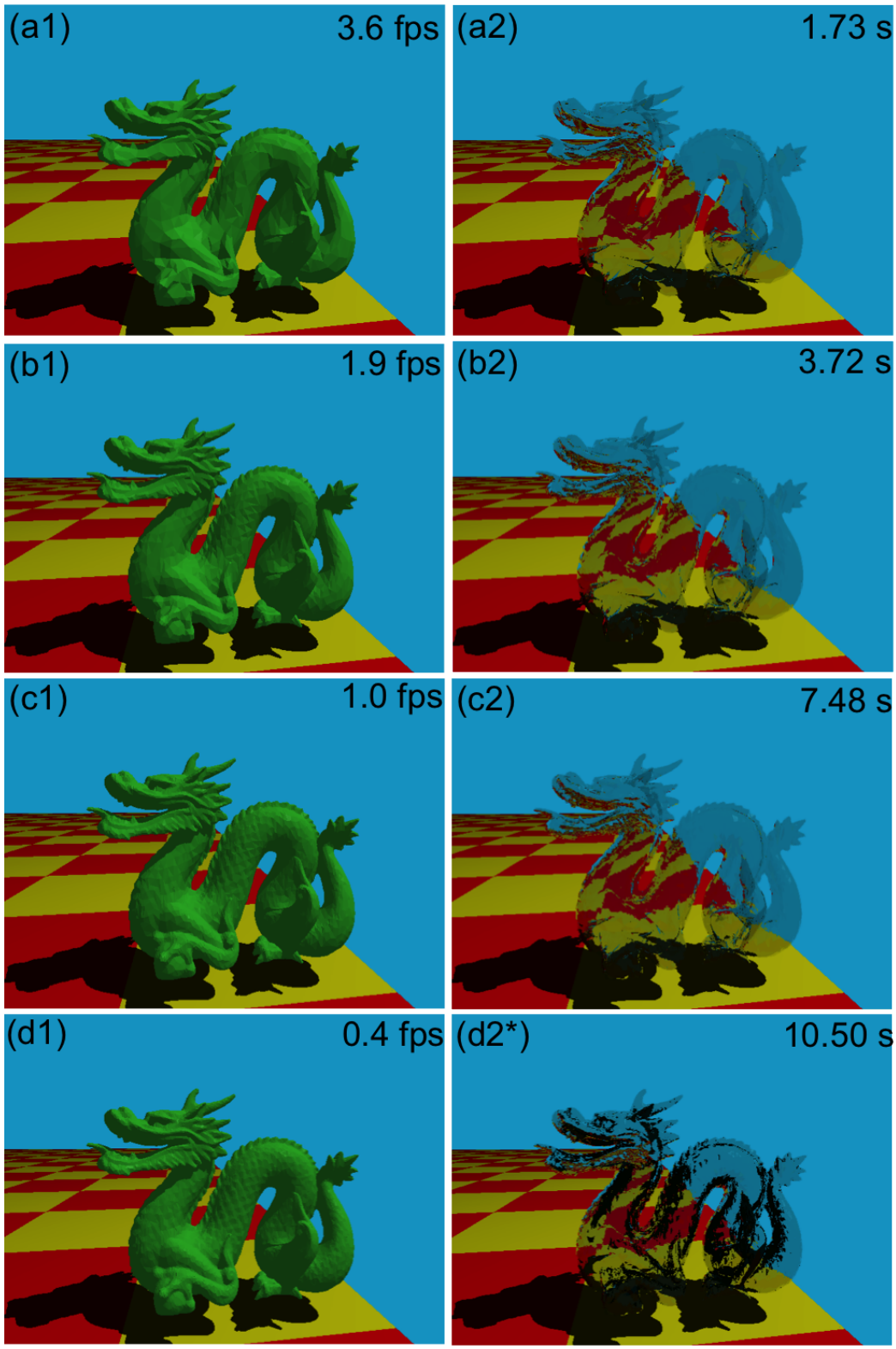


Figure 4.8: Models of the Stanford Dragon with different polygon counts, with and without secondary rays. (a) 5,000 polygons. (b) 10,000 polygons. (c) 20,000 polygons. (d) 50,000 polygons. *The secondary ray stack cannot be made large enough without exceeding the size of local memory for the 50,000 polygon model.

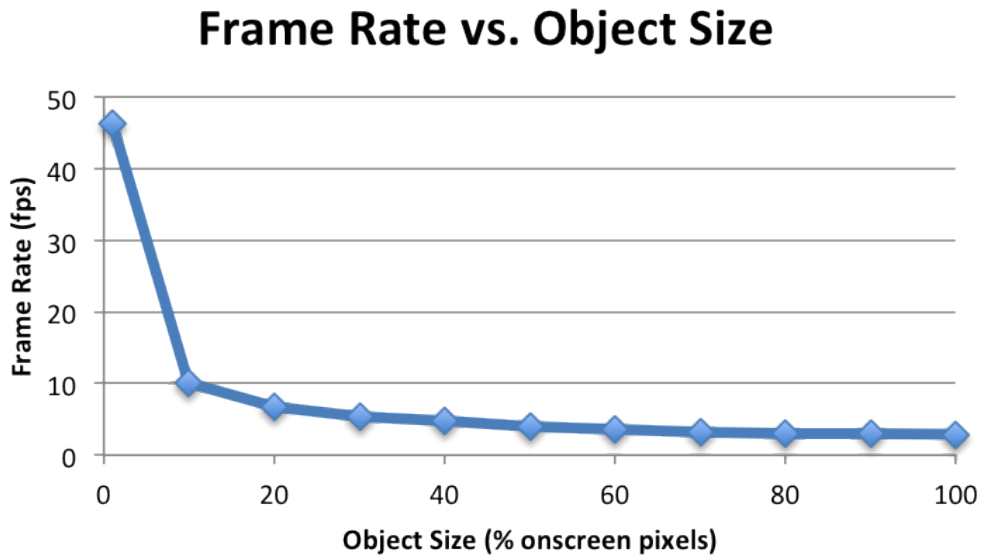


Figure 4.9: Benchmark results for varying object size from 1% to 100% of total onscreen pixels. A large performance drop occurs between 1% and 10% occupancy.

a massive performance drop. This would be very undesirable for real-world applications.

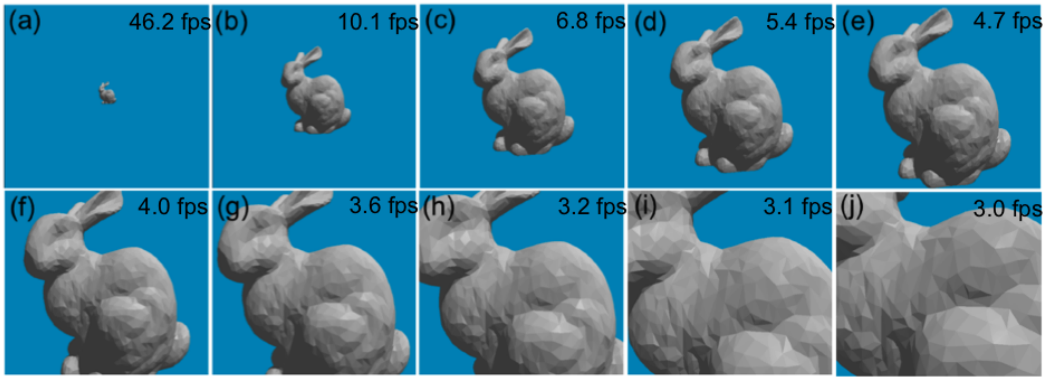


Figure 4.10: 5,000 polygon Stanford Bunny model scaled to different sizes in terms of total onscreen pixels occupied. (a) 1%. (b) 10%. (b) 20%. (b) 30%. (b) 40%. (b) 50%. (b) 60%. (b) 70%. (b) 80%. (b) 90%. 100% is not pictured.

These results indicate that, while removing all but the top level bounding sphere intersection test for most pixels is very effective, traversing the bounding volume hierarchy is a serious bottleneck. This could have a few

different causes. First, hierarchy traversal incurs a great deal of branching, even with the linearization technique that is in place in the perceptual framework. Since branching is very slow on current GPUs, a large number of branches for many OpenCL work items at once could result in a significant performance hit. There also may be a way to improve bounding volume hierarchy organization for more optimal traversal. Alternatively, this may be a common problem for GPGPU ray tracers, as object size as a percentage of onscreen pixels is not a common benchmark and is not found in existing literature.

Whatever the cause, this phenomenon explains the somewhat poor results for the polygon count benchmark compared to other tests that use a modified Whitted scene (all portions of the polygon count benchmark contain a close-up of a mesh with at least 5,000 polygons). This should be addressed in the future.

4.2.8 CPU vs GPU

One of the original goals of this work was to compare performance for CPU and GPU implementations of a perceptually optimized framework. However, as the project progressed, focus shifted away from assessment of existing techniques and toward the development of a novel technique, Refresh Rate Modulation. As such, the current framework implementation is no longer compatible with the CPU tools provided by OpenCL.

CPU vs GPU benchmarks were performed with an earlier version of the framework that featured a more traditional spatial degradation technique. The results shown in Figure 4.11 were gathered on a system with an AMD Radeon HD 6770M GPU and a 2.7 GHz Intel Core i5 CPU. A 2-level foveation was used for this test to accentuate trends, as more levels result in decreased performance. While overall performance was very poor for this earlier system, the speedup numbers do reveal an interesting performance difference for CPU and GPU architectures.

The graphical output for this benchmark is shown in Figure 4.12. The spatial degradation technique produces a significantly better speedup for the CPU, but raw GPU frame rates are much better for both full-resolution and

	Full Resolution	Foveated	Speedup
CPU	5.28 fps	93.9 fps	*17.78*
GPU	37.7 fps	222 fps	*5.89*

Figure 4.11: Performance results for an earlier version of the perceptual framework that features spatial degradation. Frame resolution is 640× 480.

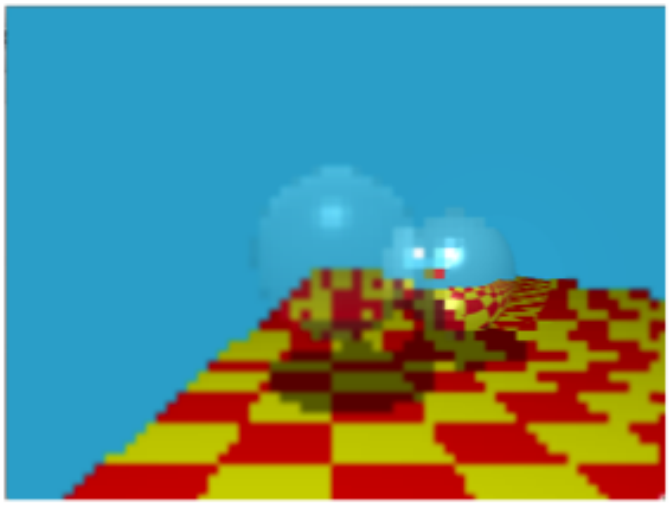


Figure 4.12: Rendered output for early framework that could be selectively executed on the CPU or GPU.

foveated output. The large CPU speedup is likely due to the fact that the CPU is much less sensitive to coherency between adjacent rays; reducing pixel density decreases workload but does not hurt computational efficiency as it does on the GPU. Pixelation is very noticeable with this approach. Adding a GLSL blur effect makes the effect less noticeable, but also significantly reduces performance.

4.3 Perceptibility Pilot Study

4.3.1 General Study

The perceptual subtlety of RRM was measured with a small pilot study. Test subjects were instructed to look at the full resolution scene. RRM was

then enabled, and subjects rated how noticeable the effect was on a scale from 1 (not noticeable) to 10 (very noticeable). Subjects moved the camera through the environment using keyboard controls to introduce motion to the scene. The eye tracker introduced an excessive degree of noise even with the smoothing filter enabled (70 pixels average error), so a mouse was used in its place for the study. Subjects were allowed to move the cursor at will, but were told to fixate on it at all times to mimic the functionality of a low-noise eye tracker.

This general study produced favorable results, with an average noticeability rating of 1.6 out of 10 for a group with 12 participants. A rating of 1 is not noticeable, whereas 10 is very noticeable. This indicates that RRM achieves an excellent computational speedup with almost no impact to the perceived quality of rendered images. A high-polygon scene with an irregular floor texture was used for the study to maximize geometric and color variety (Figure 4.13), as scene contents could affect RRM noticeability.

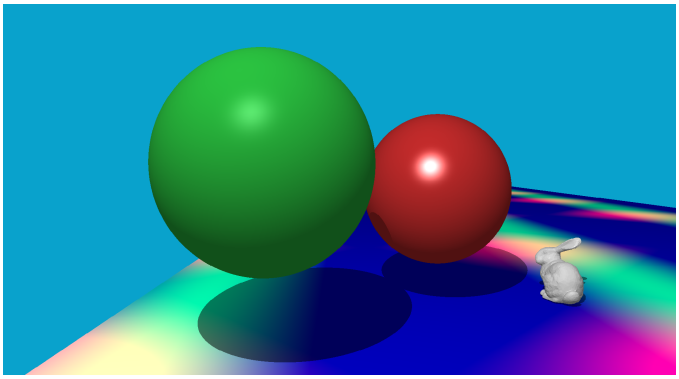


Figure 4.13: The modified high-polygon scene includes an irregular floor texture to maximize geometric and color variety.

4.3.2 Impact of Refresh Group Size on Perceptibility

After subjects in the perceptual study rated the scene for the default 3×4 refresh group dimensions, refresh group size was gradually increased and participants were asked to rate noticeability for each size. Participants indicated that the effect is very noticeable to the periphery (greater than 5 on the noticeability scale) for values of N larger than 20 (Figure 4.14). Figure

4.15 shows the high polygon scene with N = 132 to illustrate the perceptual impact of large values of N. Five users participated in this extended test.

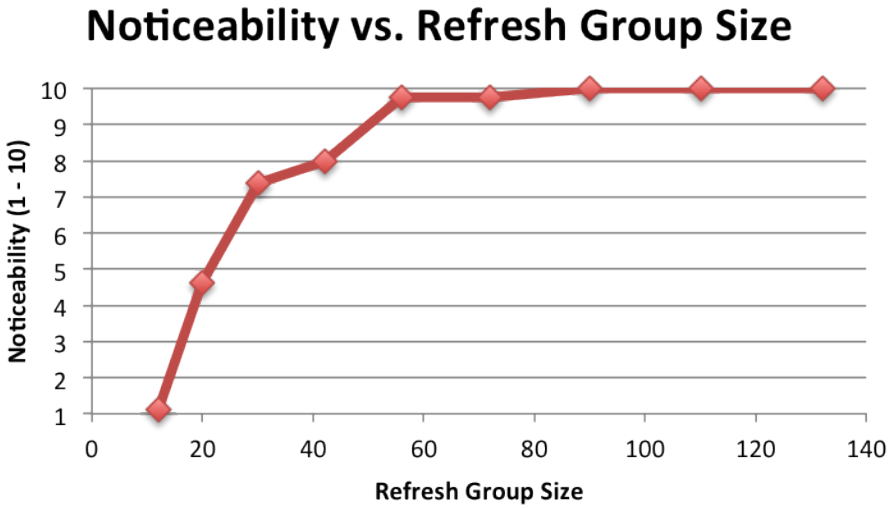


Figure 4.14: Results of noticeability study for varying refresh group size.

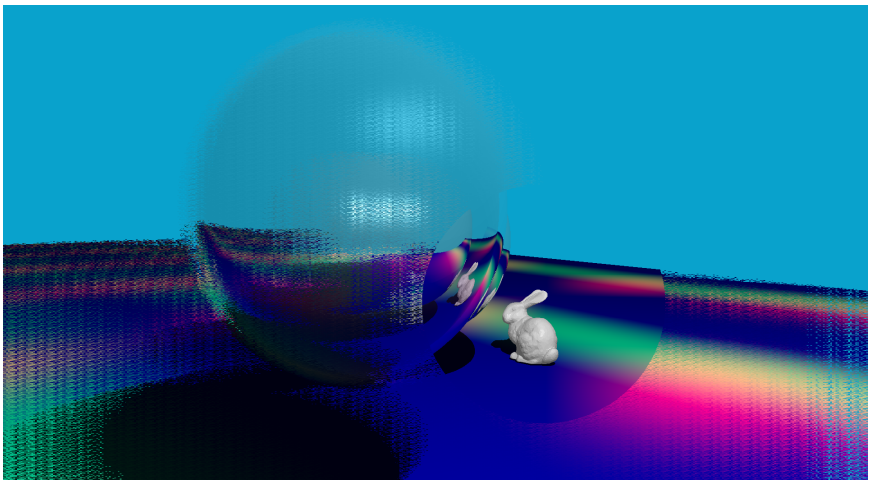


Figure 4.15: Perceptual impact of increasing the refresh group size to N = 132. The fragmentation effect is very noticeable even to peripheral vision.

4.4 Perceptually Optimized Collision Detection

A perceptually optimized collision detection system was developed to demonstrate the flexibility of the perceptual graphics system. The optimization decreases the physics calculation time for the scene shown in Figure 4.16 from 2.01ms to 1.15 ms, for a speedup of 1.78. Physics calculation times were measured over a span of 1,000 frames and averaged to ensure that the speedup is representative of overall performance.

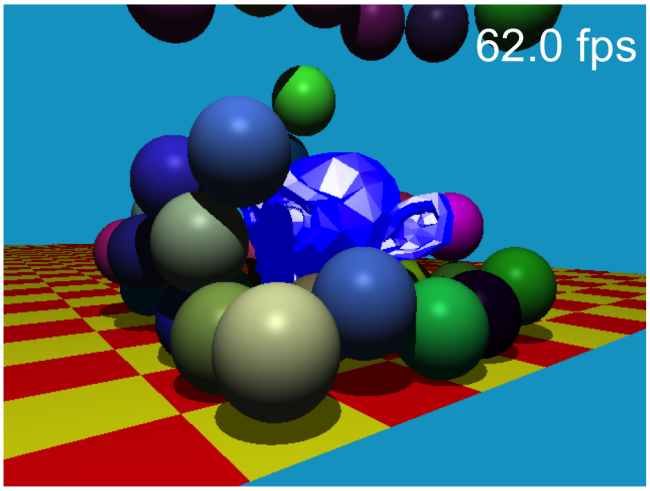


Figure 4.16: High polygon mesh with 50 moving spheres.

Chapter 5

Conclusions

Through the implementation and testing of this perceptually optimized computer graphics framework, the viability of the novel Refresh Rate Modulation (RRM) optimization technique has been established. The RRM technique partitions the viewing area into two subregions based on a model of human visual perception, and reduces the refresh rate in the outer region for up to a 3.4 speedup in performance benchmarks. It has also been shown that the framework is extensible to other perceptually optimization techniques through the incorporation of a perceptually-driven interruptible collision detection algorithm.

A pilot study indicates that RRM produces excellent computational speedup with almost no effect on perceived image quality. A more robust, full-scale user study will be performed to corroborate these findings. Additional useful data regarding ideal refresh order and the impact of scene contents on noticeability can also be gathered as part of a larger study.

Some work remains to fully realize the potential of the current system. The collision detection system can be integrated more fully with the ray-tracing engine such that the optimization is engaged via fixation instead of being manually toggled as it is now. Bounding volume hierarchy traversal can be made more efficient to allow for the close-up rendering of high-polygon models in real-time. More time can also be invested into researching the multi-pass secondary ray processing technique. A reduction routine on the GPU could be used to remove empty secondary rays in place of the CPU method that was in place for previous tests. While reduction is not well-suited to stream processors, it would avoid a number of expensive GPU-to-CPU memory operations. Finally, refresh order and refresh group

size can be adjusted in real-time to perceptually compensate for dynamic scene content. This might be accomplished through the use of visual energy functions, as described by Avidan and Shamir [3].

In the future, this framework can be extended to include a variety of perceptual optimizations. Adaptive subdivision, the simplification of polygonal meshes outside of the foveal region, is a prime candidate for this framework as it has been proven effective in providing computational speedup. Supersampling within the foveal region to improve perceived image quality while only modestly increasing computational load is another possibility. In the future, this framework will be used as a testbed for emerging perceptual graphics techniques.

Bibliography

[1] Song Ahn. OpenGL pixel buffer object (PBO), 2007. `http://www.songho.ca/opengl/gl_pbo.html`.

[2] E. Angel. *Interactive Computer Graphics: A Top-down Approach Using OpenGL*. Pearson/Addison-Wesley, 2009.

[3] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[4] K. Cater, A. Chalmers, and G. Ward. Detail to attention: exploiting visual tasks for selective rendering. In *Proceedings of the 14th Eurographics workshop on Rendering*, EGRW '03, pages 270–280, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[5] A.T. Duchowski. *Eye tracking methodology: theory and practice*. Springer, 2007.

[6] 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.

[7] Wilson S. Geisler and Jeffrey S. Perry. Variable-resolution displays for visual communication and simulation. *SID Symposium Digest of Technical Papers*, 30(1):420–423, 1999.

[8] Derek Gerstmann. OpenCL: Graphics interop, 2010. `http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/gl_sharing.html`.

[9] A.S. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Academic Press, 1989.

[10] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, May 1987.

[11] Khronos Group. gl_sharing, 2010. `http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/gl_sharing.html`.

[12] Susan M. Kolakowski and Jeff B. Pelz. Compensating for eye tracker camera movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, ETRA '06, pages 79–85, New York, NY, USA, 2006. ACM.

[13] Marc Levoy and Ross Whitaker. Gaze-directed volume rendering. *SIGGRAPH Comput. Graph.*, 24(2):217–223, February 1990.

[14] Margaret Livingstone. *Vision and Art: The Biology of Seeing*. Harry N. Abrams, Inc., 2002.

[15] G. Marmitt. *Modeling Visual Attention in VR: Measuring the Accuracy of Predicted Scanpaths*. Clemson University, 2002.

[16] S. McKee and K. Nakayama. The detection of motion in the peripheral visual field. *Vision Research*, 24(1):25–32, 1984.

[17] Ann McNamara, Katerina Mania, Marty Banks, and Christopher Healey. Perceptually-motivated graphics, visualization and 3d displays. In *ACM SIGGRAPH 2010 Courses*, SIGGRAPH '10, pages 7:1–7:159, New York, NY, USA, 2010. ACM.

[18] H. Murphy and A. T. Duchowski. Visual gaze-contingent level of detail rendering. In *Proceedings of Eurographics 2001*, Manchester, UK, September 2001.

[19] NVIDIA. *NVIDIA CUDA Programming Guide 4.2*. 2012.

[20] Carol O'Sullivan, Ralph Radach, and Steven Collins. A model of collision perception for real-time animation. In *In Proc. 1999 Conference on Computer Animation and Simulation - Eurographics Workshop (EGCAS*, pages 67–76. Springer, 1999.

[21] C. OSullivan and J. Dingliana. Real-time collision detection and response using sphere-trees. *15th Spring Conference on Computer Graphics*, pages 83–92, 1999.

[22] Alvin Raj and Ruth Rosenholtz. What your design looks like to peripheral vision. In *Proceedings of the 7th Symposium on Applied Perception in Graphics and Visualization*, APGV '10, pages 89–92, New York, NY, USA, 2010. ACM.

[23] Martin Reddy. Perceptually optimized 3d graphics. *IEEE Comput. Graph. Appl.*, 21(5):68–75, September 2001.

[24] L. Sherwood. *Human Physiology: From Cells to Systems*. Cengage Learning, 2012.

[25] Game Physics Simulation. Bullet physics library, 2012. [Online; accessed 7-May-2012].

[26] John M. Snyder and Alan H. Barr. Ray tracing complex models containing surface tessellations. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 119–128, New York, NY, USA, 1987. ACM.

[27] AMD Staff. Introduction to OpenCL Programming, 2010. [Online; accessed 8-May-2012].

[28] AMD Staff. OpenCL and the AMD APP SDK, 2011. `http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-AMD-APP-SDK.aspx`.

[29] K.G. Suffern. *Ray tracing from the ground up*. Ak Peters Series. A K Peters, 2007.

[30] W.B. Thompson, R.W. Fleming, S.H. Creem-Regehr, and J.K. Stefanucci. *Visual Perception from a Computer Graphics Perspective*. Taylor and Francis, 2011.

[31] Niels Thrane, Lars Ole Simonsen, and Advisor Peter rbk. A comparison of acceleration structures for gpu assisted ray tracing. Technical report, 2005.

[32] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–70, 2006.

[33] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[34] Wikipedia. Binary space partitioning, 2012. [Online; accessed 7-May-2012].

[35] Wikipedia. Cube mapping, 2012. [Online; accessed 8-May-2012].

[36] Wikipedia. Phong shading, 2012. [Online; accessed 7-May-2012].