Advanced lighting in 2D graphics

Ferdinand Majerech
Univerzita Pavla Jozefa Šafárika v Košiciach
3Ib, 2012-2013

Abstract: As game development costs increase and mobile gaming platforms become mainstream, there is a need for cheaper, high-performance real-time graphics. Our goal is to explore options to improve lighting in 2D graphics to provide a low-cost alternative way to create realistic graphics. The result of our work should be an advanced 2D lighting model based on lighting models normally used in 3D, a general-purpose pre-renderer generating data needed by this lighting model as well as a reference implementation of the model.

Keywords: Real-time computer graphics, 2D graphics, lighting, game development, sprite, normal map, pre-rendering, shaders

1 Introduction

Computer graphics is a field of computer science which studies manipulation and presentation of visual content. The computer game industry has very specific needs with regard to graphics; games try to create an illusion of world continuously changing in response to user input. This requires rendering a detailed scene faster than the human eye can notice.

The game industry is in an arms race to create the most realistic graphics; this has outpaced hardware development as well as software creation tools, greatly increasing development costs. AAA game budgets today are in tens of millions of dollars. Consequently, game companies avoid financial risk by reusing proven game series, limiting creativity.

Together with new funding models (e.g. alpha funding, crowd funding) and the rise of mobile devices, this has lead to a shift in the industry; many developers are abandoning large companies to start smaller projects focusing on gameplay instead of technology. These games usually have low budgets and need alternative ways to compete with high-detail graphics of AAA games. Often they need to work within constraints of mobile devices which have much less memory and processing resources than traditional platforms.

This has lead to a resurgence of use of 2D graphics, often considered dead by the mainstream companies. The subject of this paper is advancement of 2D graphics (and lighting in particular), taking advantage of modern hardware.

Creating 2D assets is considerably cheaper compared to 3D. At the same time, 2D game programming is simpler and results in lower hardware requirements; instead of 3D models with tens of thousands of triangles, we can work with sprites with two triangles (a quad). The per-object processing overhead (such as transformation) is still present, but may be simpler in 2D, depending on needs of a particular game.

As 2D removes almost all geometry processing work, it's possible to create very high detail graphics. Even 10 years ago, 2D PC games had "perfect" graphics no longer limited by hardware. In this sense, we can't improve on 24-bit RGB images; human eye won't see the difference. How good the graphics look depends on the artist, not hardware. However, lighting in 2D games is usually lacking. As there is no geometry data, it's not possible to achieve believable dynamic lighting in general case.

In this work, we augument common RGB graphics by per-pixel geometry data to enable more advanced light calculation. This should allow us to adapt existing lighting models used in 3D graphics or to create custom models taking advantage of special cases present in 2D scenes.

The main goals of this papers are as follows:

- Describe an approach to achieve believable dynamic lighting in 2D graphics.
- Implement a tool generating auxiliary data for 2D lighting from 3D data (a pre-renderer).
- Implement a dynamic lighting model working on 2D data.

An important secondary goal is to make results of our work accessible. All source code is released under an open source license (Boost), and the pre-renderer can be used by various 2D game projects.

1.1 Terminology

This section describes some graphics terms used in this article. Shaders (programmable GPU functionality) are described in more detail in a separate section.

- **GPU**: Graphics Processing Unit. Harware dedicated to graphics calculation.
- **OpenGL**: Graphics programming API supported by most gaming platforms.
- **Direct3D**: Graphics programming API used on Windows, Windows Phone and XBox consoles.
- **Sprite**: An image or an animation representing an object in a 2D game. Often has multiple *facings*, represeng a rotated object.

1.2 Shaders

In past, graphics features such as coordinate transformations, lighting and fog were implemented directly in GPU hardware, and the programmer could not extend or modify them. This is referred to as *fixed-function pipeline*, and can be used through older versions of the OpenGL and Direct3D APIs.

During the 2000s, GPUs moved to more programmable designs. Most graphics features are no longer provided by the GPU and must be implemented by the programmer in languages designed specifically to run on GPU. The most common of these are the *OpenGL Shading Language* (GLSL) used by the OpenGL and OpenGL ES APIs and *High Level Shader Language* (HLSL) used by Direct3D. Today, fixed-function GPUs are a rarity, the last major fixed-function platform being the Nintendo Wii.

Programs written in these languages are called *shaders* and run on programmable cores on the GPU. The most common of these are *vertex shaders* and *fragment shaders*. Graphics data are first processed as vertices by the vertex shader, resulting in screen position of the vertex at output. Vertices are then assembled into *primitives* (points, lines or triangles), which are *rasterized* into 2D fragments (usually pixels) forming the image on the screen. These are processed by the fragment shader to determine color of the fragment (pixel).

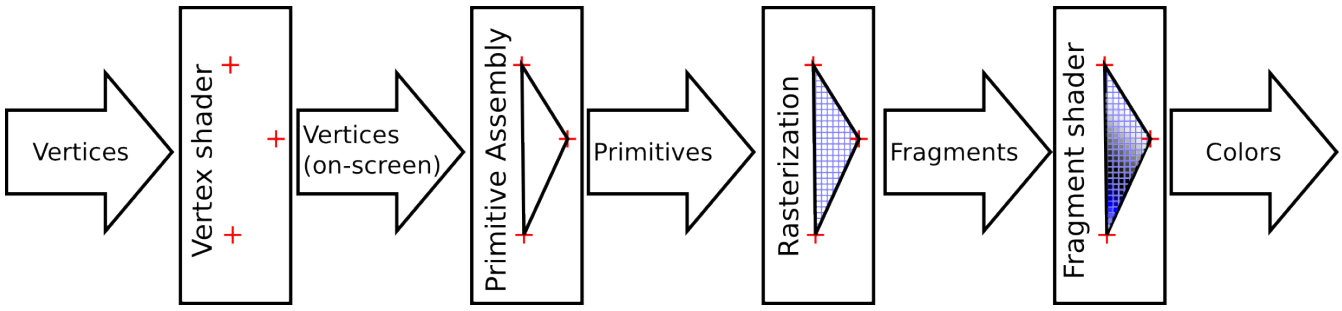

Figure 1: Basic scheme of the programmable GPU pipeline.

A vertex shader program will execute once for every vertex of graphics data processed by the GPU, while a program running on the fragment shader will execute once (or more, in special cases) per pixel of every graphics primitive. In 2D graphics, the vertex shader does little processing as there are very few vertices. The fragment shader, however still processes a large amount of data, which might be comparable to 3D games (depending on the game).

We use GLSL to write shaders, as it is supported on almost all hardware. However, our code could be easily reimplemented in other graphics languages or in a software renderer; it's not dependent on features of any specific hardware or API.

1.3 Current situation

The mainstream game industry moved away from 2D in the early 2000s. Since then, there has been little progress in 2D graphics.

Due recent market changes such as mobile devices, crowdfunding and WebGL, there are three major groups in the game industry that commonly use 2D graphics: Mobile, web, and indie PC/console developers.

The former two need 2D for its low performance requirements; indie developers need to save development costs. Recently, there've been a few attempts to innovate 2D graphics, revisiting shelved ideas from 2000s or coming with new approaches, often utilizing programmable graphics hardware.

In some 2D games, basic lighting can be achieved by exploiting specifics of the scene in a particular game. For example, a 2D game taking place in interiors can assign normals to walls and light them accordingly by doing one calculation per wall. A tiled isometric game may light objects based on the world space distance of the tile they are standing on from a light source. The first example only creates believable lighting for flat walls; the second will light an entire object homogenously. In general, these approaches don't result in believable lighting with complex objects, as there is no way to determine which parts of an object are faced towards the light, and which are faced away from the light.

Figure 2: Lighting based on tiles. Up: Antharion. Down: Tiberian Sun.

A more recent approach adapts normal mapping, a technique used in 3D graphics to increase perceived detail of surfaces. In normal mapping, a texture is used to perturb normal vectors of a surface, affecting light reflection to create an illusion of complex geometry. In 2D, normal data can be added to a sprite to specify the normal of each pixel of the sprite. This creates more realistic lighting as the parts of the object that are facing "away" from a light source are unlit and the parts facing towards the light are lit (and everything in between). This works perfectly with directional ("infinitely distant") light sources such as the Sun. A problem appears when the light source has a position. To calculate direction to the light source, world positions of pixels in the sprite must be known as well.

This can be worked around by using the 2D position of the pixel and specifying light positions in 2D. This works well for side-scrolling and top-down games, where world coordinates map trivially to the screen. It's less useful for games with different view angles, e.g. isometric games. Also, as we only have 2D coordinates, an object can never be lit from "above" or "below".

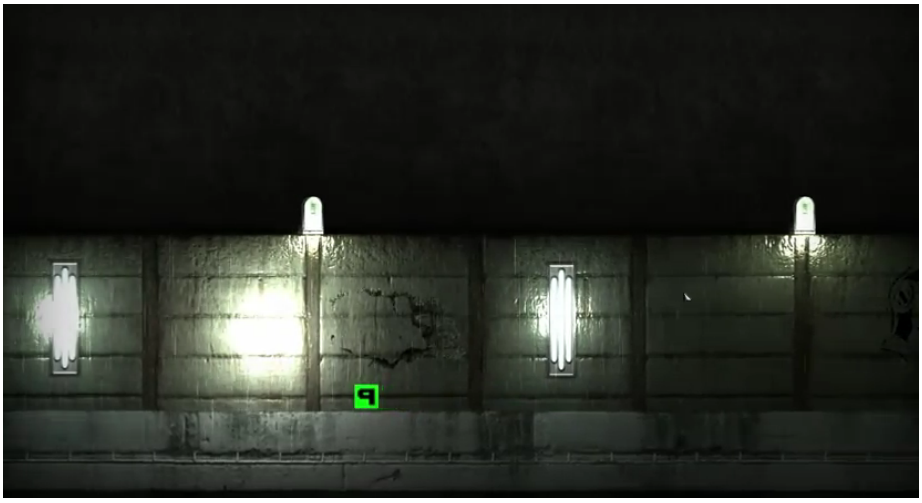

Figure 3: Normal mapped 2D lighting from a side-scrolling view. Demo by Owen Deery.

In this paper, we extend the normal mapping approach by adding another layer to the sprite, specifying 3D position of each pixel relative to the position of the object. This should allow us to get precise distances and directions between pixels on a sprite and a light source in 3D space, allowing more believable lighting. A few upcoming games, in particular, *Project Eternity* by Obsidian Entertainment and *Stasis* by Christopher Bischoff seem to be taking this approach, but unfortunately there is no open documentation or source code available.

2 Our approach

Our work can be separated into three main parts:

1. Acquire 2D data (sprites) enhanced with auxiliary data (normals, positions) needed for lighting. This is a part of graphics creation.
2. Load sprites, associate them with data specific with the current scene (such as light sources, object position) and pass them to the GPU as textures. This is done at runtime by the CPU code of a game.
3. Calculate lighting on the GPU using the sprite and scene data. This is done at runtime by shader code running on the GPU.

2.1 Sprite creation

There are various ways to create sprites for a 2D game. Besides color data, our lighting model also needs the normal and position of each pixel. Sprite creation should be as easy as possible; afterall, this is one of the main reasons to use 2D. Some ways to create sprites are liste below:

1. **Manual painting**: This is the most common way to create 2D graphics. Unfortunately it's impractical in our case as it's unintuitive to directly paint normals and positions as colors.
2. **Pre-rendering polygonal 3D data** When pre-rendering, 3D models are created just like for a 3D game, and then rendered from specific viewpoints to create a sprite. The difference is that there are no limits on vertex/face count and the model does not have to be optimized for real-time rendering. Also, only the visible parts of the model need to be created (a building might only be visible from one direction). A tool is needed to render the model into a sprite, in our case including normals and positions of pixels. All this data can be acquired from the model.
3. **Pre-rendering voxel 3D data**: Before the 3D acceleration boom, voxel (volume pixel) graphics were used in some games, and some new games are revisiting this technology. Pre-rendering voxels into 2D data is simple with current 3D hardware; they can be represented as point clouds, which is inefficient at real-time but good enough for pre-rendering. The main advantage is the simplicity of voxel creation, which resembles working with LEGO blocks, in contrast with complex 3D tools such as Maya or Blender. Accessibility of voxel creation was an important factor behind the growth of communities around games such as *Red Alert 2* and *Minecraft*. (The latter doesn't use "real" voxels, but the authoring process is the same.)
4. **Photography**: A few 2D games, such as *The Neverhood* and *Dark Oberon* use photographed objects for graphics. Recently, depth-aware devices such as Kinect and Leap Motion have reached mass market, making photography viable for our lighting model. Unfortunately, currently available

devices have very low resolution. However, this might improve in future with the next generation of depth-aware devices.

Figure 4: From left to right: Painted, pre-rendered polygonal 3D, pre-rendered voxel 3D, photography.

As the sprite creation step is separate from run time, any of these approaches can be used as long as it creates a sprite in the same format.

We chose to go with pre-rendering of polygonal data, as it's most compatible with existing developer pipelines. However, it might be interesting to explore voxels in future as they provide a very low barrier of entry for artists, and photography might become more viable as technology improves.

2.2 Pre-renderer

In game development context, pre-rendering means rendering 3D models to sprites used in a game. Usually, a model is rendered with multiple facings, used when the game object faces different directions. (South, west, northwest, etc.) Pre-rendering was common in PC and console games (especially RTS and RPG) in the 1990s and is still being used by many indie, mobile and web-based games.

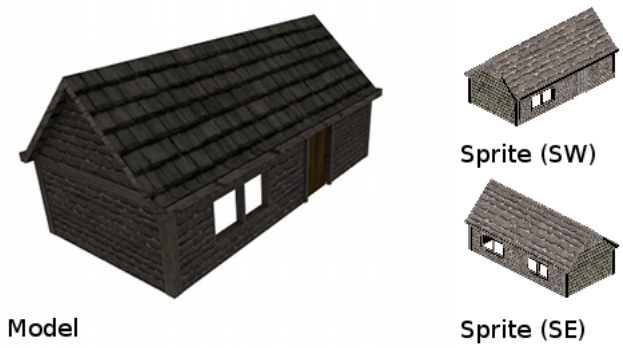

Figure 5: 3D model and two facings of a sprite created from that model.

Pre-rendering can take much longer than a real-time render in a 3D game, so it's possible to render models with very high detail; rendering times in minutes or even hours are acceptable. Rendering methods unviable in real-time are also often used, such as raytracing and radiosity.

Most game studios use in-house pre-rendering tools (*pre-renderers*). These are usually tied to whichever 3D modeling package the studio uses, and a new tool is often written for every game. Very few pre-renderers are freely available and none of these suits our needs. (In particular, we need source code access to render normals and positions for our lighting model).

We decided to implement a custom pre-renderer for our lighting model, with following goals:

- Support for multiple 3D formats, to support any modeling tool.
- Configurable for varying view angles. (Top-down, isometric, etc.)
- Support for any number of object facings. (8 are most common, some games use 16)
- Can output colors, normals and positions.
- Command-line interface to support scripting.
- Open source and gratis.

Projection. Most pre-rendered games (often incorrectly referred to as isometric) use a form of dimetric projection where the X and Y axes represent the ground plane and are scaled equally, and the Z axis is scaled separately and points upwards.

Such a projection can be achieved by rotating an orthographic projection first by 45°around the Z axis and then by α around the X axis, where α is the *view angle*, determining from how "high" we're viewing the scene. Often used

view angles are 30°(Tiberian Sun, many RTS games), 45°(Baldur's Gate, other Infinity Engine games) and 90°(top-down).

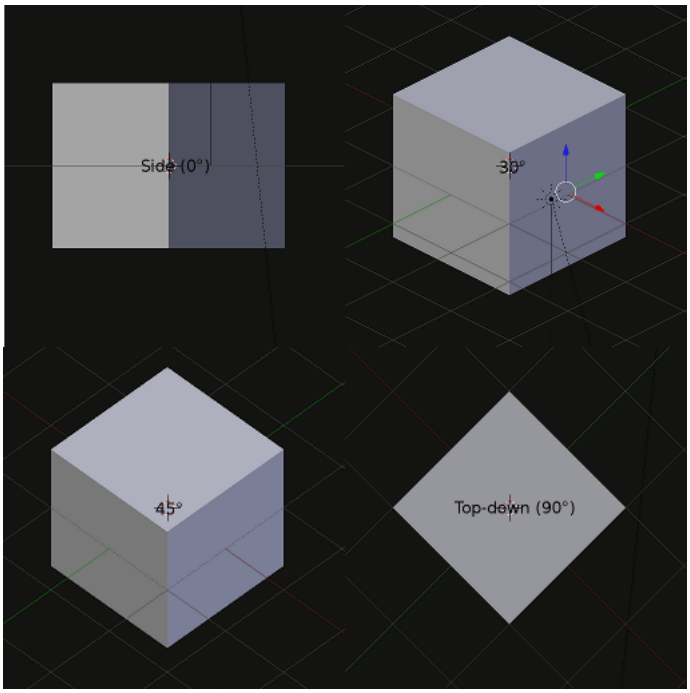

Figure 6: Commonly used view angles supported by our pre-renderer.

It should be noted that some 2D games (e.g. SimCity 4) use trimetric or even oblique projections, which can't be represented in this way.

Our lighting model must be aware of the view angle to transform object positions (which are in 3D) to 2D coordinates.

Pre-rendering process. Color, normal and position data of each facing of a sprite are separate images. We call these *layers*. In code, there are separate *Layer* classes managing rendering of colors, normals and positions; *DiffuseLayer* (diffuse color), *NormalLayer* (normals) and *OffsetLayer* (position, as offset relative to object position). The pre-renderer can be extended by more Layer classes to output different data; this might be useful for future games using different lighting models.

Work of our pre-renderer can be roughly described in following steps:

1. Process input: determine the 3D model to render, facings of the model, camera parameters, what data to render, etc.
2. Load the model.
3. For each facing:
   (a) Set up the scene. (camera, model, etc.)
   (b) For each layer:
      i. Render to texture. (drawing layer-specific data, e.g. colors or normals)
      ii. Download the texture from the GPU to an image in RAM.
      iii. Write the image to a file.
4. Write a metadata file describing each facing, camera parameters used, image filenames, etc.

Data representation. Each sprite is a set of images, but we work with data such as normals and positions. We need to represent this data as colors to store them in images.

A normal is a direction vector with three components. Its magnitude does not affect the direction, and in graphics we always store normals as unit vectors (to simplify calculation). Value of any component in a unit vector is always in the interval of **[-1, 1]** . Each channel of an RGB color is in the interval of **[0, 1]**.

The X, Y and Z components of normals are stored in the R, G and B color channels, respectively, using a simple mapping:

$$color = (coordinate + 1.0) * 0.5$$

Positions are slightly more involved, as they must be relative to origin of the object represented by the sprite. Otherwise, we couldn't move the sprite without breaking lighting. Because of this, we refer to stored positions as *offsets*. A 3D position is a 3-component vector like a normal, so we can also use an RGB color, but we need to set minimum/maximum limits for the position components so we can map to the **[0, 1]** interval. To do this, we use an axis-aligned bounding box containing the entire model.

We can then map each component to a color channel:

$$color = (coordinate - min)/(max - min)$$

Note: For texture/image storage, color channels are usually converted to integers ([**0, 255**] for 8-bit RGB color), but this is implementation dependent. Different formats allocate different numbers of bits per channel (leading to loss of information; this is usually unnoticeable with 8-bit RGB). In our pre-renderer, we work with floating-point colors until we output the result. Currently, only the 8-bit RGB output format is supported, but other formats (in particular, RGB-565) might be added later.

2.3 Graphics scene

Once we have sprites with all needed data we need to figure out how to send them to the GPU when drawing the graphics scene in a game. There are also other scene elements, such as light sources and the camera.

While our graphics scene is 2D, positions of objects and light sources must be in 3D for our light model to work. These 3D coordinates must be available to the shader where the light model is computed. Only on shader do we determine where on the screen to position the sprites.

Sprites. Each sprite consists of multiple facings representing different rotations. Each of these facings is composed of 3 textures: color, normal and offset.

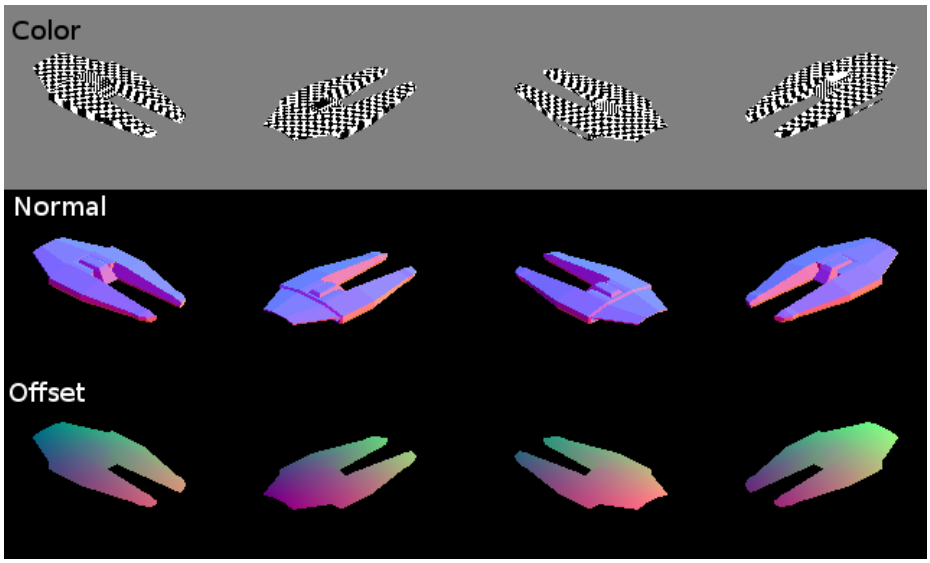

Figure 7: Layers of 4 facings of a sprite.

A major speed issue is that working with many small textures wastes both memory and execution time (due to texture rebinding). This can be solved by virtualizing texture space to place multiple sprites on the same texture.

View. Viewing a 2D scene is simple compared to 3D. However, we still need to deal with 3D object positions, which have to be transformed to 2D when drawing.

The following parameters affect the view:

- *View center* A 2D vector specifying the center of the viewport.
- *Zoom* Allows to zoom the view in/out, scaling the graphics. This is not an essential parameter, but it's useful for some types of games.
- *View angle* View angle of the projection used when pre-rendering.

The first two are used to create an orthographic projection matrix that projects 2D data to the screen. The view angle is used to recreate the projection used for pre-rendering on shader, to project the 3D object position to 2D.

Light sources. Our light model works with light sources types: *point lights* and *directional lights*. Point lights have a 3D position and can be used to model sources such as a lamp, or a fire. Directional lights are "infinitely distant"; they only have a direction towards the light. They can be used to model distant light sources such as the Sun, or a distant nuclear explosion.

A scene in a game can have a large number of light sources; for instance, in an RTS, every unit might have its own light, resulting in hundreds or thousands of lights. Calculating the contribution of each light source to every pixel would quickly become too resource-intensive. We are also limited by implementation; GLSL does not support dynamic arrays which forces us to use a fixed maximum number of lights. Currently, our scene is hardcoded to work with at most 8 point and 4 directional lights.

The next step will be to work around this limitation by managing many virtual lights (using a spatial management data structure such as an octree) and mapping the closest or most influential light sources to shader lights based on the object being drawn at the moment.

Drawing. Each game object is associated with a sprite used as its graphical representation. When drawing an object, the facing of the sprite closest to the object's rotation is chosen, and the color, normal and offset textures of that facing are bound to separate texture units.

View parameters, light sources, object position and the bounding box are passed to the GPU in form of *uniform variables* (constants from point of view of the shader program).

Finally, the sprite is rendered (as a pair of textured triangles). At that point the lighting is calculated by shaders separately on each pixel.

2.4 Lighting

Our lighting calculation runs fully on fragment shader, once per pixel. Using the pre-rendered data in the sprite and object position, we reconstruct the 3D position and normal of the pixel. This allows us to calculate lighting in 3D space.

Lighting model. Our lighting model is based on the Blinn-Phong lighting model, although we don't calculate specular lighting (this will require further pre-renderer enhancements). This means that at the moment we can't model shiny surfaces such as metals. The Blinn-Phong model provides quite realistic results without expensive computation. Its main advantage, however, is that it's the most commonly used lighting model in real-time 3D graphics, and many extensions have been developed on top of it. This should allow us to improve the model further in future without reinventing the wheel.

Our current light calculation (done separately for each color channel of the RGB color) is as follows:

$$illumintation = i_a * m_d + \sum_{l=1}^{lightCount} (a_l * m_d * i_{d,l} * directionToLight_l * normal)$$

m_d is diffuse material color. i_a is the global ambient light color, $i_{d,l}$ is diffuse color of the light, and a_l is the attenuation factor, which decreases light intensity with distance from a the source. For directional lights, the attenuation factor is 1. For point lights, it decreases with distance from the light source:

$$a_l = 1/(1 + attenuation_l * distanceToLight_l)$$

The *attenuation* parameter of a light can be used to tweak the area affected by the light. To better match reality, $distanceToLight_l$ should be squared. However, because with colors we're limited to the interval of **[0,1]**, avoiding the squaring results in more believable lighting. This could be improved by implementing HDR (High Dynamic Range) rendering in future (using the full floating-point range).

Unlike the plain Blinn-Phong model we don't have a separate ambient material color. We reuse the diffuse color as it avoids the need for even more data per pixel, and cases where ambient color differs from diffuse color are rare. If specular reflection is implemented, we will also reuse diffuse color, but multiplying it by a specular reflection value stored in another texture.

3 Conclusion

At this point, most features of the prerenderer are implemented. There are still shortcomings, however:

- It's command-line only (A separate GUI frontend should be implemented in future).
- Only single-mesh models are supported.
- Animations are not supported.
- There is no anti-aliasing.
- Every image is written to a separate file; there are no spritesheets.

All of these features will have to be implemented in future; however, that will most likely not be covered by the bachelor's thesis.

A basic lighting demo is also implemented, with diffuse lighting based on the Phong model. However, due to hardware limitations, only a fixed number of lights is supported.

The main priority now is implementation of specular lighting. After that, depending on time constraints, the focus will be on light source virtualization (to work around the fixed light count) and possibly extensions to the light model such as self-shadowing and HDR.

There are also many options for further work; e.g. ways to represent prerendered data more efficiently and alternative 2D lighting models.

References

1. Bui Tuong Phong. *Illumination for computer generated pictures.* Communications of ACM 18, no. 6, 311-317 (June 1975)
2. James F. Blinn. *Models of light reflection for computer synthesized pictures.* SIGGRAPH Comput. Graph. 11, 2, 192-198 (July 1977)
3. Blinn, J. F. 1978. *Simulation of wrinkled surfaces.* SIGGRAPH 1978.
4. A. Cignoni, C. Montani, R. Scopigno, and C. Rocchini. *A general method for preserving attribute values on simplified meshes.* In Proceedings of the conference on Visualization '98 (VIS '98). IEEE Computer Society Press, Los Alamitos, CA, USA, 59-66 (1998)