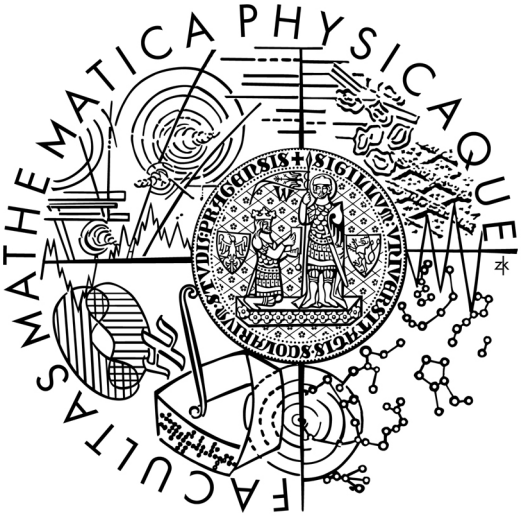
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Štěpán Vondrák

Nefotorealistické zobrazování v reálném čase

Kabinet software a výuky informatiky

Vedoucí diplomové práce: **Doc. Ing. Jiří Žára, CSc.**

Studijní program: **Informatika**

Rád bych poděkoval vedoucímu diplomové práce Doc. Ing. Jiřímu Žárovi, CSc. za poskytnutí studijních materiálů a Ing. Danielu Sýkorovi za cenné rady a připomínky. Děkuji své rodině, přátelům a kolegům za jejich podporu, trpělivost a porozumění.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 16. dubna 2006 Štěpán Vondrák

Contents

Název práce: Nefotorealistické zobrazování v reálném čase
Autor: Štěpán Vondrák
Katedra: Kabinet software a výuky informatiky
Vedoucí diplomové práce: Doc. Ing. Jiří Žára, CSc.
e-mail vedoucího: zara@fel.cvut.cz
Abstrakt: Cílem této práce je prostudovat metody nefotorealistického zobrazování (NPR) s ohledem na jejich použití při zobrazování v reálnem čase a využít možností moderních grafických karet k implementaci několika NPR technik. První část srovnává implementaci tří hranových detektorů, které fungují na principu hledání nespojitostí v obrazech s normálami, druhými derivacemi hloubek či identifikátory jednotlivých regionů vykreslované scény. Společným použitím těchto detektorů lze nálézt siluety objektů, okrajové hrany, ostré hrany a rozhraní mezi materiály. Kombinace zvýrazňování důležitých hran a stupňovitého stínovaní povrchu objektů je použita k zobrazení scény ve stylu kreslených filmů. Závěrečná část práce se věnuje inovativnímu přístupu k zobrazovní trojrozměrných modelů ve stylu mozaik. Ten oproti dříve prezentovaným metodám dokáže vykreslovat i složité scény v reálném čase. Hlavního urychlení je docíleno tím, že k výpočtu pozic jednotlivých dlaždic je namísto tradičního Voroného diagramu použita simulace sytému tuhých pružin.
Klíčová slova: počítačová grafika, nefotorealistické zobrazování, hranové detektory, kreslený film, mozaiky

Title: Real-time non-photorealistic rendering
Author: Štěpán Vondrák
Department: Department of Software and Computer Science Education
Supervisor: Doc. Ing. Jiří Žára, CSc.
Supervisor's e-mail: zara@fel.cvut.cz
Abstract: The goal of this thesis is to examine non-photorealistic rendering (NPR) methods with respect to their application in real-time rendering and to utilize the features of modern programmable graphics processing units (GPUs) to implement several NPR techniques. To achieve maximal efficiency of the implemented rendering styles, extra care is taken to offload all geometry processing from the CPU to the GPU. Implementations of three image-space edge detectors are presented and compared. Silhouettes, border edges, creases and material boundaries can be identified by detecting discontinuities in an image with world-space normals, in the second derivative of the depth buffer and in an image with region identifiers. Combination of the edge detectors and a stepped shader is used to render objects in a cartoon style. A novel approach to rendering of a three-dimensional scene in mosaic style is proposed. Unlike previously presented methods, the implemented technique can render complex scenes at interactive framerates. To achieve real-time performance, a simulation of a system of infinitely stiff springs is used instead of centroidal Voronoi diagrams.
Keywords: computer graphics, non-photorealistic rendering, edge detection, cartoon, mosaics

Chapter 1

Introduction

As the performance of computers, video cards and game consoles increases, the complexity and realism of movie visual effects, computer-rendered movies and video games dramatically rises. However, as the rendered graphics gets closer to portraying the reality, people start to wonder whether we are also getting closer to the *Uncanny Valley*.

The Uncanny Valley is a term conceived by Masahiro Mori [26] in 1970. It is a principle of robotics concerning the emotional response of humans to robots and other non-human entities. It states that as a robot is made more humanlike, the emotional response from a human to the robot will become increasingly positive, until a point is reached at which the robot's appearance becomes strongly repulsive.

While many disagreed and criticized this theory, lately it has become very popular especially in regard to computer animation characters. For example, the box-office failure of Square Pictures' movie Final Fantasy: The Spirit Within is often cited as an example of the Uncanny Valley — the characters portrayed in the movie look very realistic, utilizing complex skin and hair shaders and cloth physics simulation, yet they don't move and behave like real humans and are perceived as alien of just "strange". On the other hand, most characters in Pixar movies are more stylized or even "cartoony" and appeal to a much wider audience.

In the past, when fast consumer 3-D graphics accelerators became available, most people expected the games to have as realistic graphics as possible and the developers strived for realism. Advances in video card capabilities lead to corresponding advances in graphics engines — from textured polygons with static precomputed lighting to complex scenes with dynamic lighting and shadows, normal-mapped or parallax-mapped surfaces or shaders simulating water surfaces. With the next generation of GPUs and game consoles, there is a risk that the Uncanny Valley might become a problem in the game industry too.

Recently, perhaps to prevent a fall into the valley, or just to differentiate their products from the mass of "realistic" games, many game developers started to employ a wide range of non-photorealistic visual styles.

This thesis focuses on the real-time non-photorealistic rendering of three-dimensional polygonal scenes. The goal is to implement NPR techniques that utilize the modern programmable GPUs to achieve interactive framerates. Extra care is taken to do as little scene and image processing on the CPU. This limitation is very important for real-time performance, especially when scenes with hundredths of thousands polygons are rendered. Any CPU scene processing or heavy communication between the processor and the video card, such as image uploading and downloading, can become a major bottleneck in the rendering pipeline. Because the speed of the GPUs rises much faster than the speed of CPUs, the gap widens each year. Also, in interactive three-dimensional applications, the CPU is needed to simulate the avatar's movement in the virtual world, the world's physics or the AI of the computer-controlled avatars, therefore it is always helpful to offload the scene rendering to the GPU.

The rest of the thesis is organized as follows. First, the framework of the NPRView application which was used to develop, test and tweak the implemented NPR techniques is described. Next, the individual NPR techniques are introduced and their real-time implementation is presented and evaluated.

Chapter 2

NPRView framework

In this chapter the core framework of the NPRView application is described. The framework shares some traits with NVIDIA's FX Composer, but its goals are different.

The primary purpose of FX Composer is to provide an IDE for easy shader development and debugging. It allows a user to assign material shaders to individual objects and post-processing shaders to the scene and to tweak the parameters of the shaders.

The NPRView framework is designed to allow an easy way to prototype, test and implement new rendering techniques. Each technique consists of a sequence of one or more rendering passes. When compared to FX Composer, a rendering pass is roughly equivalent to rendering the scene using a specified material shader or to applying a post-processing shader to the scene. But the framework allows implementation of any kind of passes and is not limited just to these two cases. The parameters controlling the behavior of the passes are exposed in the user interface and can be modified at run-time.

2.1 Framework overview

The framework allows a user to define any number rendering *techniques*. Some of the implemented techniques include a cartoon shader or a mosaic renderer. But there are also techniques that do not display a scene in a non-photorealistic rendering style, instead they can be used to debug, tweak or fine-tune Cg programs or the parameters of rendering components. For example, several implemented techniques can be used to present how the implemented edge detectors work and compare their results.

Techniques consist of one or more rendering *passes* which are processed in a sequential order. Each pass utilizes a single rendering *component* that implements its functionality. The pass provides initial values of the component's *variables*, which control the component's behavior.

At run-time, the user can switch between all implemented techniques, tweak

the values of the component variables of all passes and immediately view the results.

Program 2.1 shows an example of a simple depth-based screenspace edge detector technique. Figure 2.1 on page 9 illustrates how the example technique works.

```
# First pass named phong_and_depth uses component render_phong to draw the scene.

pass render_phong phong_and_depth {
    render_target "phong" ; # Render the scene into a texture instead of framebuffer.
} ;

# Second pass named edge_detect detects edges in depth texture generated in pass 1.

pass edge_detect_depth_2nd_derivative edge_detect {
    texture "<generated>:phong_depth" ;
} ;
```

Program 2.1: Simple edge detector technique.

The technique consists of two passes. The name of the first pass is `phong_and_depth`. The name is optional and is used only to distinguish individual passes in the user interface. The pass uses component `render_phong` to render the scene into two textures. The names of the two textures are derived from the value of the `render_target` component variable, which is "phong" in this case; the color buffer will be rendered into a texture named `"<generated>:phong_color"`, and the depth buffer will be rendered into a texture named `"<generated>:phong_depth"`. Texture name syntax is explained in subsection 2.2.2 on page 12.

The second pass is called `edge_detect` and utilizes the `edge_detect_depth_2nd_derivative` component to detect edges in the depth texture specified in its `texture` variable. In the example, the depth buffer rendered in the first pass is used to detect the edges. Since no render-target is specified in the second pass, the resulting image is rendered into the framebuffer.

The following list summarizes the basic NPRView framework terminology:

Technique A sequence of rendering *passes*.

Pass A rendering pass specifies which *component* should be used to render the pass as well as initial values of the *component's variables*. The example technique has 2 passes named `phong_and_depth` and `edge_detect`.

Component A component is a C++ object that implements a single rendering pass. In the example, two components are used: `render_phong` draws currently selected scene using the Phong shading and `edge_detect_depth_2nd_derivative` detects edges in the provided texture.

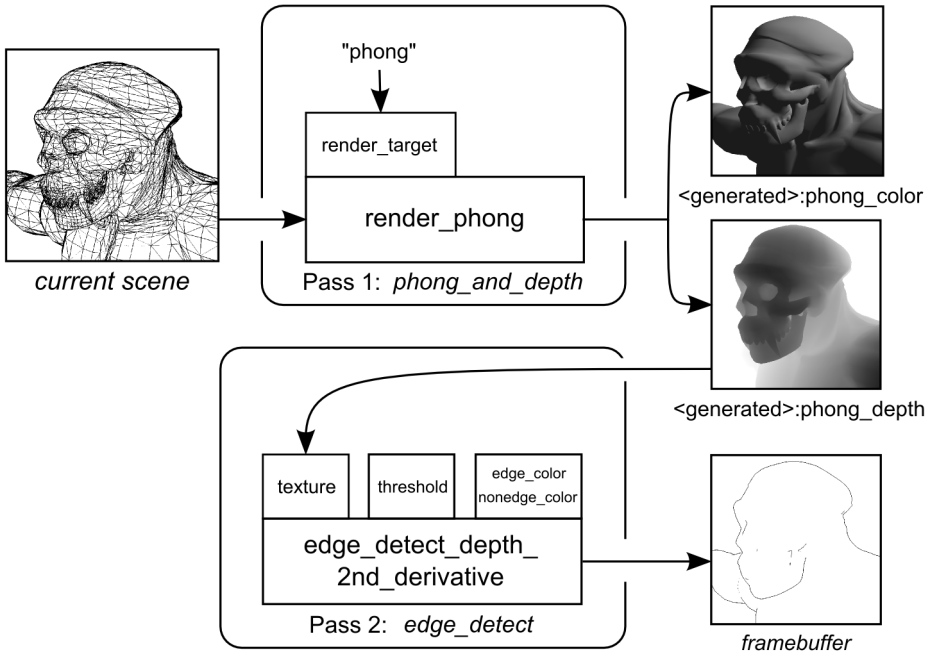


Figure 2.1: Simple edge detector technique.

The behavior of each component is controlled by its *component variables*. The initial values of the variables are specified in the pass and selected variables may be tweaked by a user at run-time.

Component variable A typed variable that is used to specify component setup. Component variables have additional properties such as a range (minimum and maximum) or a default value.

Component descriptor Describes the interface to the component. Descriptors list all variables of each implemented component, and specify properties of the variables.

The rest of this chapter describes components and techniques in more detail.

2.2 Components

Components are the core building stones of rendering techniques. All components are implemented as C++ objects inheriting the `Component` class.

Several components are implemented in NPRView. They can be divided into two groups:

- Generic components that rely on Cg programs specified as their parameters to do the hard work. For example, the `simple_render` component draws

current scene using vertex and fragment programs specified by its component variables.

- Technique-specific components which are utilized by technique passes that can not be easily implemented using a generic component. For example, the `draw_tiles` component generates mosaic tile geometry from information about the tile positions and orientations provided in its input textures.

2.2.1 Component descriptors

For each component, a corresponding *component descriptor* defines the interface to the component. The descriptor lists the *component variables* of the component it describes, and provides information about the types of the variables, their limits or user interface properties.

Component descriptors are stored in external textual files and are loaded at the NPRView startup, which allows component customization without recompiling the application. More complex component customization can be accomplished by *component descriptor inheritance*, described later.

At technique load-time, when a pass definition is loaded, the descriptor of the component is used to parse the initial values of the component's variables, and instantiate the corresponding component. At run-time, the descriptor is used by the NPRView user interface to allow a user to modify the component variables.

Program listing 2.2 on page 13 shows the component descriptor of a component `simple_render`. It is used as an example of various features in the latter sections of this chapter.

Figure 2.2 on page 11 demonstrates how the `simple_render` component can be used within the framework. The component object instance is displayed in the middle of the picture. It has two input sources:

- Implicit inputs (on the left of the picture) are data that are always read by the component implementation. In this case, the component renders the current scene, so the scene is the only implicit input.

- The component behavior is controlled by its *component variables*. The interface to the variables is defined by the component descriptor (shown directly above the component), the initial values are defined in the pass that uses the component, and some of the variables may be modified at run-time in the NPRView user interface.

The component renders its output either to the framebuffer (when no render target is specified), or to the specified render target textures (which can be further processed in the following passes), as shown at the bottom of the picture.

Figure 2.3 shows how the component variables of a pass using the `simple_render` component can be modified at run-time, and how the resulting color outputs of the pass are affected by such changes.

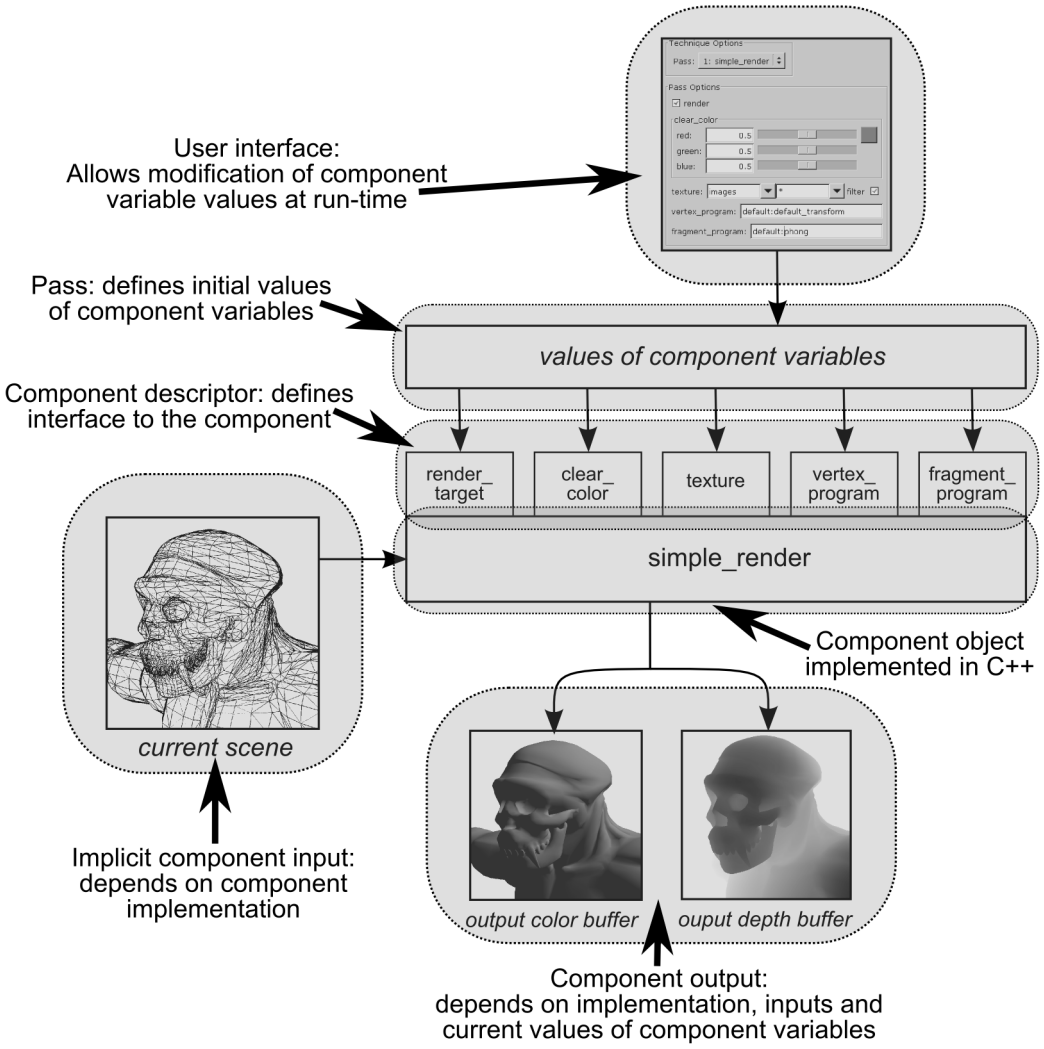


Figure 2.2: Component simple_render and surrounding framework.

The most important component variables of `simple_render` are `vertex_program` and `fragment_program`, which specify the Cg program to use when rendering the scene. By using different programs, the component can be used to produce vastly different results. And since both techniques (and therefore all their passes) and Cg programs are loaded from external files at NPRView startup, it is possible to implement new techniques without the need to change and recompile the NPRView application; only component implementations are written in C++. The figure shows two different fragment programs. One draws the scene using the Phong shading, the other one draw world-space surface normals.

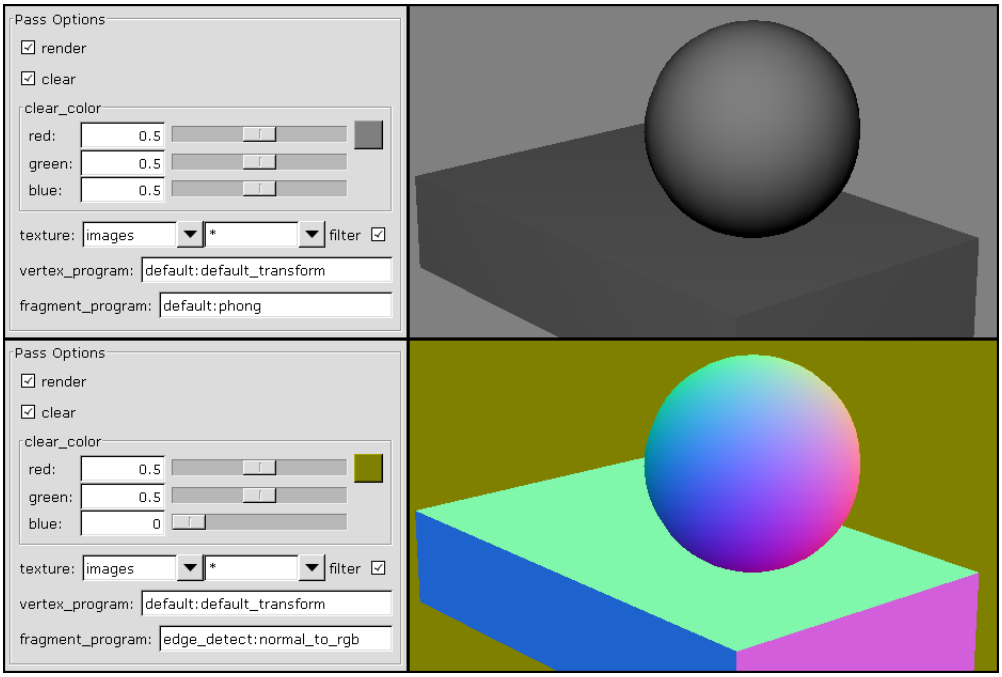


Figure 2.3: Changing simple_render setup at run-time.

2.2.2 Component variables

Component variables provide an easy way to tweak component parameters. First, variable types and properties are described in a component descriptor as shown in program listing 2.2 on page 13. Next, initial variable values can be changed per-pass in technique definition as illustrated in program listing 2.1 on page 8. Finally, variable values may be tweaked at run-time in NPRView application, unless a component descriptor hides them from the user interface.

The following component variable types are supported:

bool A boolean variable, with either *false* or *true* value. Value can be also spec-

```
component simple_render {
    # Render-to-texture variables.

    string render_target {
        default "" ;   # No render-target texture - render to framebuffer by default.
        internal 1 ;   # This hides the variable from the user interface.
    } ;
    int render_target_width {
        default 0 ;
        minimum -64 ;
        maximum 1024 ;
        internal 1 ;
    } ;
    int render_target_height {
        default 0 ;
        minimum -64 ;
        maximum 1024 ;
        internal 1 ;
    } ;

    # Color to use when clearing the framebuffer.

    rgb clear_color {
        default 0.5 0.5 0.5 ;     # Default color - grey.
        minimum 0.0 0.0 0.0 ; # "Minimum" color - black.
        maximum 1.0 1.0 1.0 ; # "Maximum" color - white.
    } ;

    # Optional texture to apply to meshes in the scene.

    texture texture {
        default "images:*" ; # Use the first available image in the "images" directory.
    } ;

    # Names of vertex and fragment programs to use to render the scene.
    # These strings must be specified in the pass definition (or when inheriting
    # component descriptor). The specified programs will be used to render the scene.
    # By using different programs, the simple_render component may be used to
    # render the scene in vastly different ways.

    string vertex_program ;
    string fragment_program ;
} ;
```

Program 2.2: Component descriptor of simple_render.

ified as an integer, a zero value being equal to *false*, a non-zero value to *true*.

int A 32-bit signed integer.

number A 32-bit floating-point number.

string A string value. When a string is interpreted as a Cg program name, its value is expected to have *file:function* format, where *file* is the base file name of a Cg file to load (without an extension or a path), and *function* is the name of a vertex or a fragment function to call. For example, `"edge_detect:normal_to_rgb"` refers to program entry function `normal_to_rgb()` read from file `edge_detect.cg`.

vector2 A two-component vector of 32-bit floating-point numbers. In textual files (component descriptors or technique definitions), vector values are stored as individual component values separated by a whitespace.

vector3 A three-component vector.

vector4 A four-component vector.

rgb A three-component color (red, green, blue). Colors are stored in the same way as vectors, with the standard value range of individual color components being 0.0–1.0.

As an example of a *rgb* variable descriptor, the descriptor of the `simple_render` on page 13 defines variable `clear_color`, with allowed range from black to pure white, and grey as a default color.

rgba A four-component color (red, green, blue, alpha). When interpreted as a transparency value, an alpha equal to 0.0 corresponds to a transparent color, and an alpha equal to 1.0 corresponds to an opaque color.

texture A texture. Texture variable values use *directory:image:filter* syntax, where *directory* is a directory name relative to one of the configured image source directories, *image* is a base image file name without an extension, and *filter* is a 0/1 integral value specifying whether the trilinear (when the value is equal to 1) or the nearest (when the value is equal to 0) filtering should be used when the texture is sampled.

When an asterisk is used as *image*, the first available image in the *directory* is used. If the *filter* specification is omitted, it is interpreted as `:1`.

The special *directory* `<generated>` contains textures generated by components. The filter specification of generated textures is ignored, the nearest sampling is always used.

Several examples of texture variable values were already mentioned. In program listing 2.1 on page 8, `<generated>:phong_depth` is used as the source

texture of the edge detector component. In program listing 2.2 on page 13, the variable `texture` defaults to `images:*`, i.e. the first available image in the `images` directory.

A variable type and name specification can be optionally followed by a definition of component variable properties, as shown in the `simple_render` example. The following properties are supported:

default *value* Defines the default value of the variable. The default is used when a pass does not specify any initial value for the variable. If no default is defined in the variable descriptor, the initial value of the variable must be set in each pass using the component.

minimum *value* The minimal value of the variable. Extreme values can not be set for variables of type `bool`, `string` or `texture`. When a value outside of the specified range is set to the variable, a warning is printed and the value is clamped to the range. Extremes for multi-component variable types (vectors and colors) are interpreted component-wise.

maximum *value* The maximal value of the variable.

ui_minimum *value* The user interface minimal value of the variable, also called a "soft minimum". It is used to limit the ranges of user interface sliders. A value outside the soft range can still be set to the variable in the numeric edit box control. Note that setting a *minimum* also sets the corresponding *ui_minimum* to the same value automatically, so definition of an *ui_minimum* is needed only if the user interface slider limits should be more strict than the "hard" limits.

ui_maximum *value* The user-interface maximal value of the variable, also called "soft maximum". Note that setting a *maximum* also sets the corresponding *ui_maximum* to the same value automatically.

vertex_parameter *boolean-value* If set to `true`, the value of the variable will be automatically applied to the corresponding uniform parameter of the Cg vertex program used by the component. This parameter, along with `fragment_parameter`, provides a way to bind any component variable to a Cg program uniform, and thus allow a user to tweak the uniform parameter at run-time.

Both `vertex_parameter` and `fragment_parameter` are silently ignored by components that do not use a Cg vertex or fragment program, respectively.

fragment_parameter *boolean-value* If set to `true`, the value of the variable will be automatically applied to the corresponding uniform parameter of the Cg fragment program used by the component.

read_only ***boolean-value*** Marks the variable as read-only in the user interface. The variable and its value will be visible in the NPRView application, but it will not be possible to change the value.

internal ***boolean-value*** Hides the variable from the user interface.

2.2.3 Component descriptor inheritance

Sometimes it is useful to define multiple components which use the same core C++ object, but have slightly different component descriptors. Component descriptor inheritance serves that purpose — it allows definition of a new component by modifying the component descriptor of an already existing component. The core component implementation remains the same, only the interface changes.

Neither component used in the technique defined in program listing 2.1 on page 8 is a *core component*[1], both `render_phong` and `edge_detect_depth_2nd_derivative` use component descriptor inheritance.

For example, the `render_phong` component is created by inheriting `simple_render` descriptor. Definition of the `simple_render` component descriptor was already shown in program listing 2.2 on page 13. The descriptor of the `render_phong` component specifies defaults for vertex and fragment program variables and hides them from the user interface, as shown in program listing 2.3 on page 17. It also overrides the default value of the clear color, and hides the `texture` variable from the user interface, as the Phong shader fragment program does not use a texture.

Figure 2.4 illustrates this component descriptor inheritance. The `render_phong` component uses the same core C++ component implementation object as the `simple_render` component (`SimpleRender`), but has a different descriptor (interface to the component).

2.3 Implementing new core component

In many cases, the components that are already implemented in NPRView are generic and powerful enough to implement new rendering techniques. However, in same cases, a more complicated technique may require a special component to implement one or more of its passes.

This section describes how to implement a new core component. The implementation of the `copy_to_texture` component is used as an example. This component is rather simple; it copies the contents of the framebuffer (either the color or the depth buffer) to a specified texture, as illustrated in figure 2.5. The implicit input of the component is the current framebuffer contents. The source buffer (color or depth) is specified by the boolean variable `depth`, and the name of the output texture is read from the string variable `texture`. The output texture

[1] Components implemented as C++ objects, such as `simple_render` are referred to as core components, in contrast to components that are defined using component descriptor inheritance.

```
component render_phong : simple_render {
    # The "modify" keyword can be used to modify properties of
    # an already defined component variable.

    modify vertex_program {
        default "default:default_transform" ;
        internal 1 ;
    } ;
    modify fragment_program {
        default "default:phong" ;
        internal 1 ;
    } ;
    modify clear_color {
        default 0.5 0.5 0.5 ;
    } ;
    modify texture {
        internal 1 ;
    } ;
} ;
```

Program 2.3: Component descriptor of render_phong.

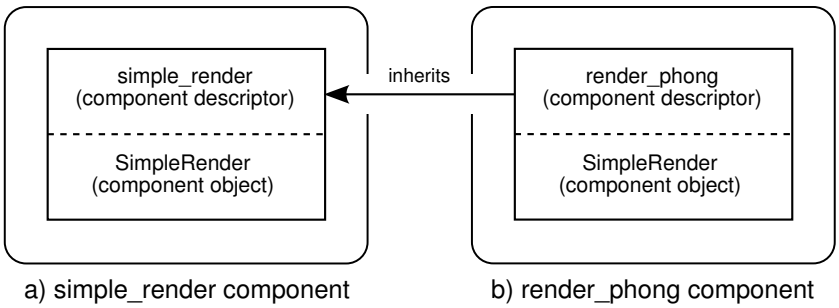


Figure 2.4: Component descriptor inheritance. The `render_phong` component inherits and modifies the descriptor of `simple_render`, but the core implementation (`SimpleRender`) remains the same.

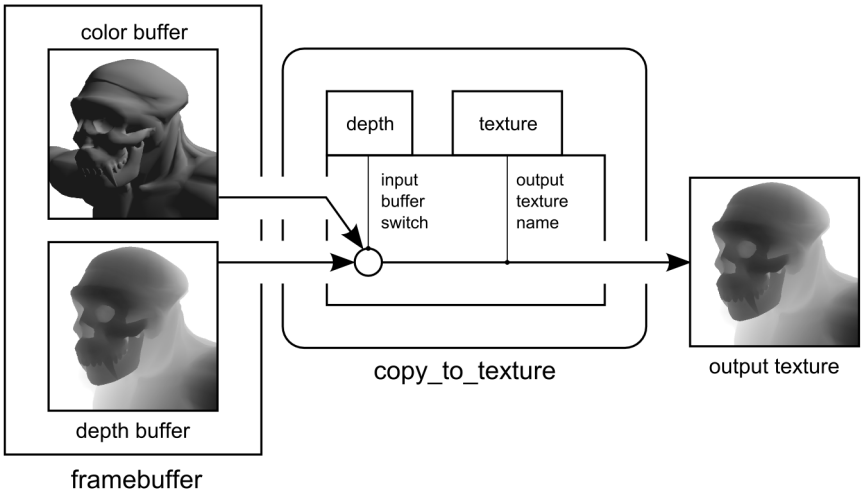


Figure 2.5: The copy_to_texture component. The framebuffer always consists of the color and depth buffer in NPRView. The component reads one of them depending on the value of the `depth` variable, and copies the contents to a texture specified by the `texture` variable.

is automatically stored in the `<generated>` directory, i.e. the complete texture name will be `<generated>:`*`texture`*.

Some unimportant source parts are omitted from the code listings, the complete sources are available in the `src/components` directory.

To implement a new component, the following steps must be performed:

1. Implement a C++ class inherited from the `Component` class. Each pass using the new component creates an instance of the class, and when such pass is rendered, the `render()` method of the component object is called.
2. Write the component descriptor of the component.
3. Register the component to the component manager.

2.3.1 Component interface

The basic features of the base `Component` class can be seen in program listing 2.4 on page 19.

Each component class must have its component descriptor and component creation function registered to the *component manager* as explained in section 2.3.4. The type of the creation function is defined as `Component::CreateFn`. When the definition of a pass is read from a technique file, the component manager calls the component creation function with a reference to the pass as its only argument. The component instance keeps the reference to the pass in `pass`.

```
class Component {
    /// A function that creates a component.
    typedef Component * CreateFn( const Pass & pass ) ;

    /// Reference to pass using this component.
    const Pass & pass ;

    /// "render" boolean variable.
    const ComponentVariable & v_render ;

    virtual void render( RenderContext & render_context ) = 0 ;
} ;
```

Program 2.4: `Component` interface.

In addition to other information about the pass, each `Pass` object has a container mapping component variable names to their values. As a convenience, most components keep references to all component variable values they're interested in.[2] In this case, `v_render` references a boolean variable that specifies whether the pass should be rendered (when `true`), or the rendering of the pass should be skipped (when `false`). This variable is automatically available to all components and does not have to be defined in a component descriptor, although it may be modified using the `modify` keyword, to override its default value or to hide it from the user interface, for example.

The core functionality of each component is implemented by the `render()` method. It is called once by each technique pass when the technique is rendered. The `render_context` object, received as the method argument, provides information about the current scene, camera settings and the repository of all available Cg programs.

2.3.2 Component implementation

Each core component class must inherit the `Component` class, define the component's name as a static member string `name`, define a creation static method `create()` with type `Component::CreateFn`, and override and implement the `render()` method. The definition of the `CopyToTexture` class that implements the `copy_to_texture` component is shown in program 2.5 on page 20.

The example component also has references to both of its component variables. `v_texture` and `v_depth`, and a pointer to the destination texture object (`texture`).

Program listing 2.6 contains the implementation of the component.

The meaning of component name and creation function was already explained, and their definitions are pretty straightforward.

[2] While referencing a container members is usually not a good idea in C++, it is allowed here because the pass structures never change once they're initialized.

```
class CopyToTexture : public Component {
    /// Component name and creation function.
    static const std::string name ;
    static Component * create( const Pass & pass ) ;

    /// Target texture name, without the 'generated' directory part.
    const ComponentVariable & v_texture ;

    /// Copy depth instead of rgba?
    const ComponentVariable & v_depth ;

    /// The texture itself.
    ManagedTexture * texture ;

    CopyToTexture( const Pass & pass ) ;
    virtual ~CopyToTexture() ;
    virtual void render( RenderContext & render_context ) ;
} ;
```

Program 2.5: `CopyToTexture` interface.

The constructor calls the base class (`Component`) constructor first, which stores the `pass` reference in the component object, and initializes a reference to the standard `render` component variable. Then, references to the both component variables are extracted from the pass, and the destination texture is created.

All texture objects are reference counted — the texture object returned by the `create_generated_texture()` method of the texture manager might be shared by multiple components. Therefore the destructor can't delete the texture object directly. Instead, it just removes its reference. The texture manager can then automatically destroy unreferenced texture objects.[3]

The `CopyToTexture::render()` method first checks if the pass rendering should be skipped.[4] Then it changes the size and the format of the destination texture if required, and copies the framebuffer contents into the texture.

The `copy_to_texture` component example is rather simple, since it does not use the more complex features of the render manager (access to current scene or Cg program handling) and the texture manager. These features are used in other core components, their implementation is available in the `src/components` directory.

[3] The texture object is not destroyed immediately when its reference count drops to zero to prevent unnecessary reloading of texture images from the disk.

[4] The `Component::should_skip()` method simply returns negation of the `render` component variable, referenced by `v_render`.

```
/// Component name.
const std::string CopyToTexture::name = "copy_to_texture" ;

/// Create CopyToTexture object.
Component * CopyToTexture::create( const Pass & pass )
{
    return new CopyToTexture( pass ) ;
}

/// Constructor.                                                    10
CopyToTexture::CopyToTexture( const Pass & pass )
    : Component( pass )
    , v_texture( pass.get_variable( "texture" ) )
    , v_depth( pass.get_variable( "depth" ) )
{
    // Create the texture to ensure it's already available to other passes.

    texture = &get_global_texture_manager().create_generated_texture( ... ) ;
}
                                                                    20
/// Destructor.
CopyToTexture::~CopyToTexture()
{
    if ( texture ) { texture->remove_reference() ; }
}

/// Render component - copy framebuffer contents to a texture.
void CopyToTexture::render( RenderContext & render_context )
{
    // Skip the pass if requested.                                  30

    if ( should_skip() ) { return ; }

    // Get framebuffer sizes and ensure the texture has correct format and dimensions.

    unsigned width, height ;
    render_context.get_framebuffer_sizes( width, height ) ;
    texture->initialize_generated( get_texture_format(), width, height ) ;

    // Copy framebuffer contents to the texture.                    40

    gl::Texture * const gl_texture = texture->get_texture() ;
    gl_texture->bind() ;
    glCopyTexSubImage2D( gl_texture->get_target(), 0, 0, 0, 0, 0, width, height ) ;
}
```

Program 2.6: `CopyToTexture` implementation.

2.3.3 Component descriptor

As already mentioned, each component, including a core one, requires a component descriptor. All descriptors are loaded on the NPRView application startup, and they're used to parse pass definitions and to initialize the user interface.

Descriptors of all core components are stored in file `core.component` in directory `data/components`. Listing of the `copy_to_texture` component descriptor is shown in program 2.7 on page 22.

Note that the `texture` component variable is of type `string`, not `texture`, since it contains the name of the image to generate. On the other hand, variables of type `texture` contain a texture object directly, and are usually used when a component uses the texture as an input.

Both `texture` and `depth` variables are also marked as internal, which hides them from the user interface, because this component can not handle change of these parameters at run-time. This is a limitation of the component itself, not the framework. It would be easily possible to rewrite the component to handle modifications of the variables by reinitialing the destination texture object when needed. For example, the `generate_texture` and `ping_pong` components support such changes to their output and temporary texture objects, respectively.

```
component copy_to_texture {
    # Texture to copy the framebuffer contents to.

    string texture { read_only 1 ; } ;

    # If false, copies the color buffer, if true, copies the depth buffer.

    bool depth { default false ; read_only 1 ; } ;
} ;
```

Program 2.7: `copy_to_texture` component descriptor.

2.3.4 Component registration

Finally, it is important to let the framework know about the newly implemented core component. All components are registered at the NPRView application startup in the `initialize_render()` method defined in file `src/nprview/init.cpp`.

Chapter 3

Drawing image edges

For a long time, many drawing styles have heavily relied on highlighting the important features of the scene by accenting specific lines in the image. Lines have been used both in artistic drawing or non-photorealistic rendering (comics, pencil or pen-and-ink drawings) and technical or medical illustrations. When rendering polygonal meshes, these lines correspond to polygon edges, and can be classified into the following categories:

Silhouette Edges shared by a front-facing and a back-facing polygons.

Countour The subset of a silhouette that separates an object from the background.

Border edge An edge adjacent to only one polygon.

Crease An edge between two polygons whose dihedral angle is greater than a specified threshold.

Material boundary An edge between two polygons with a different material (texture).

Artistic edges Edges manually tagged by an artist that should be always drawn.

3.1 Previous work

Since line drawing is integral to a lot of rendering styles, many algorithms for line rendering have been developed. A very thorough overview of various line detection algorithms and line rendering styles is provided by Isenberg et al.[15] The methods can be divided into two distinct groups: object-space and image-space algorithms.

Object-space methods extract information about important edges from the mesh polygons and then draw the edges using a variety of edge rendering styles. Polygonal meshes can be preprocessed to detect crease and artistic edges, but silhouette edges are view-dependent and can change from frame to frame, therefore the mesh must be processed regularly to detect them.

The trivial methods process all edges of the mesh at each frame and classify all edges shared by front- and back-facing polygons as silhouettes. Several data structures and algorithms have been proposed to speed-up the mesh processing, for example an *edge buffer* by Buchanan and Sousa [1]. Freudenberg et al. [8] reduce the number of potential silhouette edge candidates by analyzing only strictly convex smooth edges. Other rendering techniques use probabilistic methods and interframe coherence to approximate the silhouette, for example Markosian et al. [23]

Image-space algorithms process an image produced by rendering the scene, and detect discontinuities in the image to extract edge pixels. The result is a new image with the same resolution. It can either classify the pixels as edges and non-edges (the result is a binary image), or contain edge intensities.

Edges usually can not be successfully detected by processing the color buffer that is the result of a photorealistic rendering method, such as Phong shading with texturing. Saito and Takahashi [29] detect discontinuities in the depth buffer (silhouettes) and in the first derivative of the depth buffer (creases). Decaudin [4] proposes to render world-space normals at each pixel by drawing all meshes as pure-white with special lighting in several passes, and then classifies discontinuities in the normal image as edges. Mitchell [25] was the first to implement combination of these techniques on a GPU.

Unlike object-space edge detection, image-space edge rendering methods map to current programmable video cards very well. Fragment programs are flexible enough to render various information about each scene pixel into the color buffer (for example, scene world-space normals), and then process the rendered image (for example, detecting discontinuities in the normal image). Since the output of image-space algorithms is a texture, no information is provided about the edge geometry. Therefore they are not suitable for artistic or stylistic rendering of the edges, such as the one presented by Sousa and Prusinkiewicz [32].

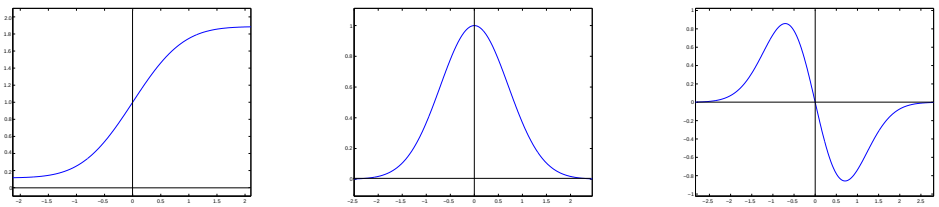
3.2 Image-space edge detectors

Image-space edge detection is a long-researched problem and a lot of different algorithms have been developed over the history. The detectors differ in their mathematical and algorithmic properties, some of them are tailored to a specific task.

This section provides an overview of several basic step edge detectors. A much more detailed overview of image-space edge detectors is provided by Ziou and Tabbone [33]. Heath et al. [14] performed a thorough comparison and evaluation of several well-known edge detectors.

A step edge in an intensity image can be detected either as a maximum in the first derivative (gradient) of the image, or as a zero crossing in the second derivative of the image, as shown in figure 3.1

The gradients in horizontal and vertical directions (G_x and G_y) of a two-



(a) Image intensity (b) First derivative (c) Second derivative

Figure 3.1: An edge in 1D function.

dimensional image can be approximated by applying the Sobel operator to the image. The operator is implemented as a pair of $3 * 3$ convolution kernels shown in figures 3.2(a) and 3.2(b). Edges can then be detected by thresholding the magnitude of the gradient, defined as:

$$|G| = \sqrt{{G_x}^2 + {G_y}^2} \tag{3.1}$$

Usually, the magnitude is approximated by

$$|G| = |G_x| + |G_y| \tag{3.2}$$

which can be computed much faster.

Also, the orientation of the edge can be computed as

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.3}$$

+1	0	−1
+2	0	−2
+1	0	−1

(a) Sobel (horizontal)

−1	−2	−1
0	0	0
+1	+2	+1

(b) Sobel (vertical)

0	1	0
1	−4	1
0	1	0

(c) Laplacian

Figure 3.2: Sobel and Laplacian convolution kernels.

To measure the second derivative of an image, the Laplacian of the image can be used. The Laplacian $L(x, y)$ of intensity image $I(x, y)$ is defined as:

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \tag{3.4}$$

The Laplacian of a discrete image can be approximated by convolution with kernel shown in figure 3.2(c). The edges in an image can be then located by searching for zero crossings in its Laplacian.

Approximation of a second derivative measurement on the image is very sensitive to noise. The Gaussian smoothing operator is usually applied to the image to suppress the noise before the Laplacian is computed. In two dimensions, the Gaussian has the form:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.5}$$

where σ is the standard deviation of the distribution.

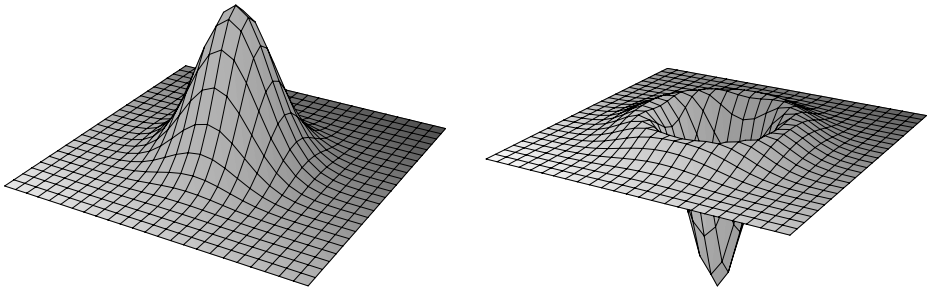


Figure 3.3: Gaussian and Laplacian of Gaussian.

Since the convolution operation is associative, the Gaussian smoothing filter can be convolved with the Laplacian filter to get the "Laplacian of Gaussian" (LoG). When LoG is convolved with the image, it smooths it and computes the Laplacian at the same time. The two-dimensional LoG function with Gaussian standard deviation σ has the form:

$$L{\circ}G(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.6}$$

The two-dimensional Gaussian and Laplacian of Gaussian functions are shown in figure 3.3.

These generic edge detectors are usually used on color or monochromatic images, for example photos, which can be very noisy, distorted, bad-lit, out-of-focus, or contain various objects with distinctive textures. It is hard to define which parts of such images should be classified as edges, and even harder to conceive an algorithm that detects the edges. The more complex edge detectors try to deal with these problems as well as possible.

In this regard, detecting edges in a rendered three-dimensional scene is a much easier problem, as many kinds of information about the scene (for example pixel normals or depths) can be extracted and then processed by a specialized edge detector. Also, there is no noise in a rendered image, therefore no complicated smoothing is necessary.

3.3 Implemented edge rendering methods

Edge detecting techniques implemented in NPRView use a combination of several image-space methods. Edges are detected in images with the following pixel properties:

Normals Discontinuities in the dot products of the world-space normals of adjacent pixels are classified as edges.

Depths Discontinuities in the first or the second derivative of the depth buffer are classified as edges.

Region identifiers Edges are reported at boundaries between pixels with different region identifiers.

When combined, the methods are able to outline object silhouettes, creases and folds, and material boundaries.

All implemented techniques consist of two or more passes. In the first pass, the scene is rendered into a texture using a specific Cg fragment program. The fragment program outputs information about each scene pixel depending on the selected edge detector, for example pixel depths or normals. In the subsequent pass, the generated texture is processed by the edge detector and a binary image distinguishing edge and non-edge pixels is drawn. Optionally, more passes that further modify the image may follow. This is illustrated in figure 3.4.

This approach requires no scene preprocessing at all. This is a very important characteristic, as any processing of the scene on the CPU could hurt the performance a lot, especially with the scene sizes the latest GPUs can handle. It also allows us to store all static scene data on the GPU to conserve computer memory.

The implemented texture-processing edge detectors (pass 2 in figure 3.4) sample a cross-shaped neighborhood of a texel to decide whether there is an edge at the texel or not, as depicted in figure 3.5.

Modern GPUs contain a dedicated hardware that can interpolate up to 8 vector variables between vertices. These vectors are usually used for interpolated colors (when using Gouraud shading), texturing coordinates or normals. The edge detectors use this feature to simplify fragment programs — the texturing coordinates for all 5 samples in the cross-shaped pattern are computed by a vertex program at each corner of the processed texture, and the hardware automatically interpolates the coordinates for each processed texel. This is illustrated in figure 3.6. Basically, this is equivalent to passing the same texture into the fragment program five times — once at its original position, and four times shifted by one texel upwards, downwards, to the left and to the right.

3.3.1 Detecting edges as discontinuities in surface normals

The first edge detector analyzes the surface world-space normals at each pixel of the rendered scene. The normal-based edge detector was first introduced by De-

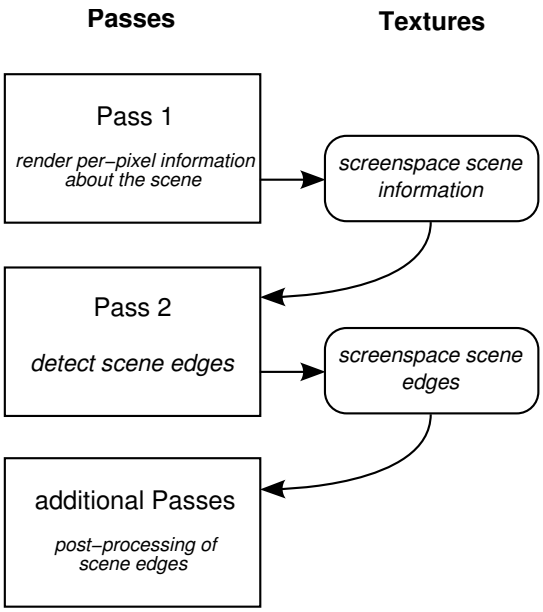


Figure 3.4: Generic edge rendering.

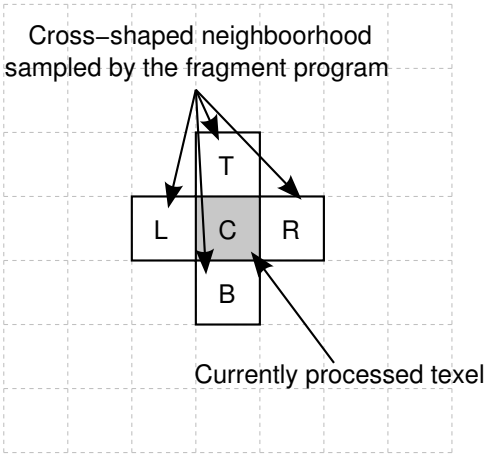


Figure 3.5: Sampling pattern used by edge detector fragment programs.

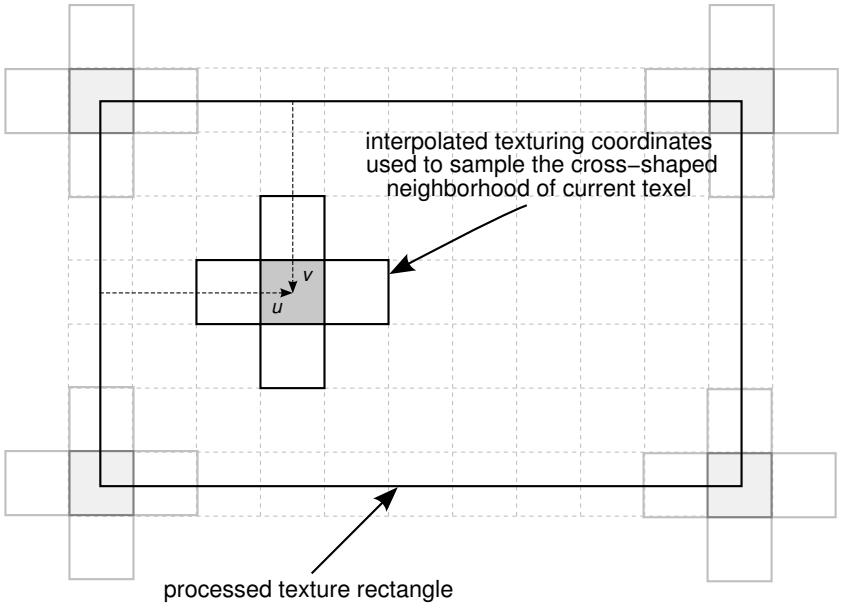


Figure 3.6: The cross-shaped texturing coordinates are computed per-vertex, and the dedicated hardware interpolators are used to determine the per-fragment uv coordinates.

caudin [4, 5]. The first real-time implementation using GeForce 3 class hardware register combiners was proposed by Dominé et al. [7]. Mitchell [25] combined this approach with depth-based and region id-based edge detector implemented in DirectX Pixel Shaders 1.4. This approach was refined by Nienhaus and Döllner [27].

The method detects discontinuities in image containing per-pixel world-space normals and is able to find most silhouette edges, border edges and creases. In NPRView, the edge detector is implemented as two passes, illustrated in figure 3.7.

In the first pass, a fragment program is used to render the scene's world-space normals into a RGB $(nx_{world}, ny_{world}, nz_{world})$ texture. Since NPRView does not use floating-point color buffers[1], the x, y and z components of the normals must be transformed into $\langle 0, 1\rangle$ range before storing them in the color buffer. The background pixels are filled with a color corresponding to the normal pointing into the screen.

In the second pass, the generated texture is processed using a discontinuity filter implemented as a fragment program. First, normals at each texel in the cross-shaped pattern are determined by expanding the texel RGB colors back to

[1]Floating-point color buffers were first introduced with the GeForce 6 videocard series, but their implementation lacked some necessary features, such as texture filtering and blending. The GeForce 7 series remedy these deficiencies, but the floating-point buffers require higher memory bandwidth, and therefore are slower to process. The precision of eight bits per component is good enough for normal-based edge detector.

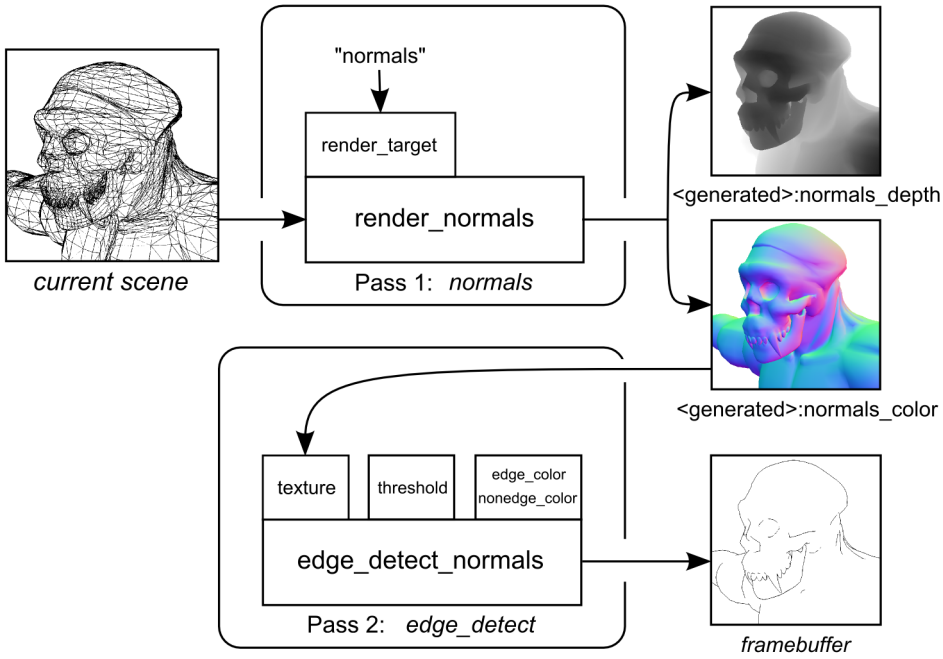


Figure 3.7: Detecting edges in world-space normals image.

$\langle -1, 1 \rangle$ range. Next, a minimum of dot products between the center texel and its neighbors is determined by:

$$d = \min\{\mathbf{n}_C \cdot \mathbf{n}_L, \mathbf{n}_C \cdot \mathbf{n}_R, \mathbf{n}_C \cdot \mathbf{n}_B, \mathbf{n}_C \cdot \mathbf{n}_T\} \tag{3.7}$$

where $\mathbf{n}_X$ denotes expanded world-space normal at texel X.

The dot products correspond to cosines of the angles between the center and neighbor normals. The minimum dot product corresponds to the largest discontinuity. If d is lower than the specified threshold, the texel is classified as an edge. Figure 3.8 illustrates the two passes in more detail.

Alternatively, the thresholding can be turned off by setting the component variable `apply_threshold` of the second pass to `false`. In that case, the detector outputs color obtained by linearly interpolating between non-edge and colors using factor f computed as:

$$f = t \cdot \big(4 - (\mathbf{n}_C \cdot \mathbf{n}_L + \mathbf{n}_C \cdot \mathbf{n}_R + \mathbf{n}_C \cdot \mathbf{n}_B + \mathbf{n}_C \cdot \mathbf{n}_T)\big) \tag{3.8}$$

where t is the value of the `threshold` component variable, which is used as an edge intensity multiplication factor when the thresholding is turned off. This equation yields a zero interpolation factor (which corresponds to the non-edge color) when all surfaces in the cross-shaped neighborhood are coplanar.

This method detects the following edge types:

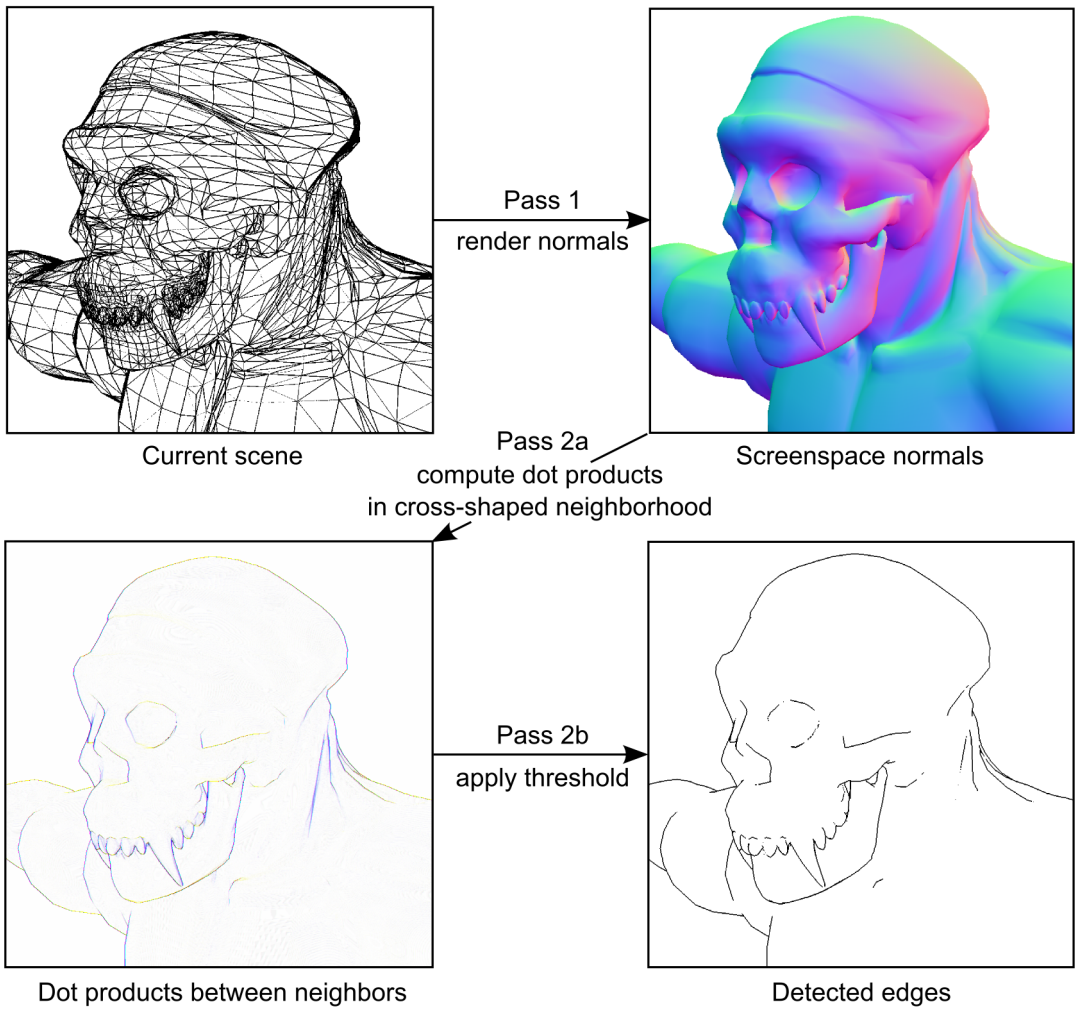


Figure 3.8: Normal-based edge detector.

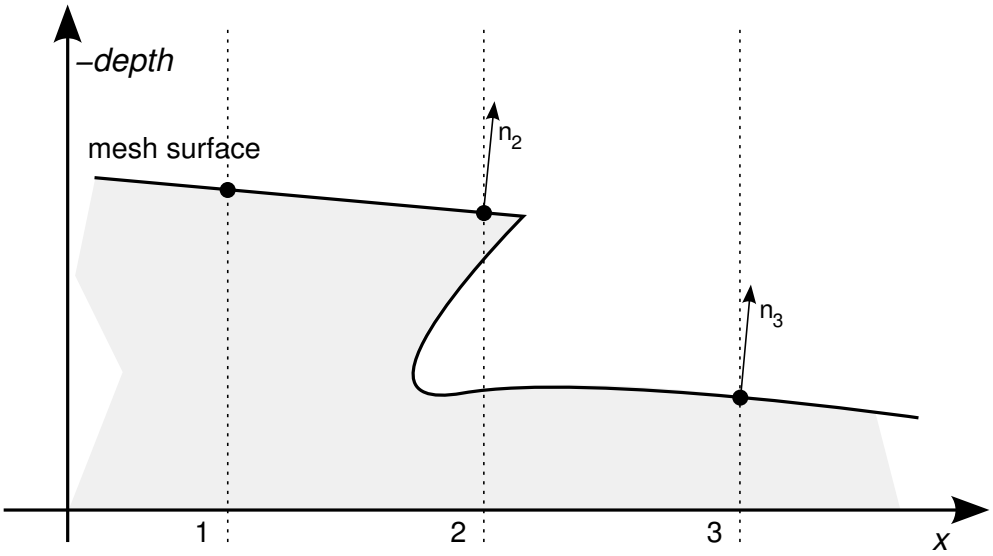


Figure 3.9: Situation where a silhouette edge can't be detected from world-space normals.

- Creases with intensities corresponding to specified threshold.
- Contour edges. This is ensured by filling the background with color corresponding to back-facing a normal. Because all rendered pixels belonging to an object have front-facing normals, the dot product between such pixels and the background pixels always passes the threshold test.
- Most silhouette and border edges.

3.3.2 Detecting edges in depth image

Not all silhouette and border edges correspond to discontinuities in normals, as illustrated in figure 3.9. While the normals n_2 and n_3 are nearly parallel, there is an obvious silhouette edge between the corresponding samples.

Takahashi and Saito [29] use the depth buffer to detect edges in image-space. Decaudin [4] combines this approach with normal-based edge detector. First real-time implementation of this method was proposed by Mitchell.[25]

Since the normals generated in the first pass are stored only in RGB channels of the generated texture, Mitchell [25] uses the alpha channel of the texture to store per-pixel depths. This approach has several drawback. First, the 8-bits of the alpha channel usually do not provide enough precision for generic scenes, even when the near and far clipping planes are chosen very carefully. There are two basic options how the proposed method can be improved to obtain a better precision: we could either use the depth buffer (modern graphics hardware can

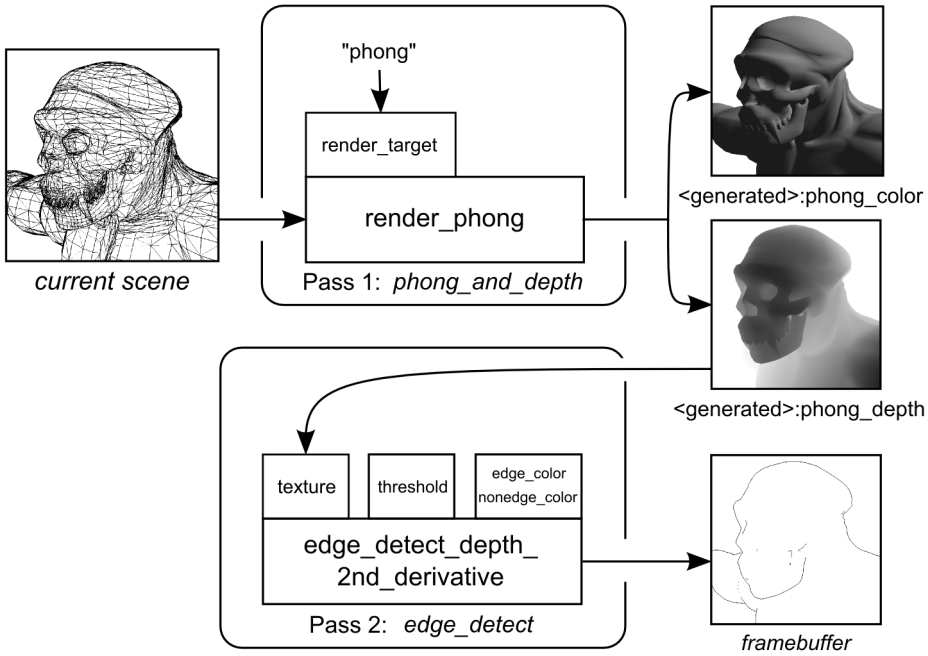


Figure 3.10: Detecting edges in depth buffer.

render the depth buffer into 16-bit or 24-bit textures), or we could render per-pixel depths into a high-precision texture. Neither option is flawless — the hardware depth buffer does not store linear depths (its precision decreases with distance), and hardware support for high-precision textures is limited in OpenGL.[2]

Two depth-based edge detectors are implemented in NPRView, both use the former option — the hardware depth buffer. The first method detects discontinuities in the depth buffer, the second one detects discontinuities in the depth buffer derivative. Figure 3.10 illustrates the second method. The depth buffer is generated whenever a scene is rendered, regardless of the applied fragment program, so any scene-rendering pass can be used as the first one. The figure uses the Phong shader as an example, but the output from `render_normals` could be also used.

As mentioned, the values stored in the depth buffer are not linear distances, and therefore must be transformed before the edge detector processes them. The transformation from the eye-space (where the eye-space linear distance is still available) to the window coordinates (the depth values stored in the depth buffer) is described in section "Coordinate Transformations" of the OpenGL 2.0 specification [30]. First, the projection from the eye-space to clip coordinates is applied

[2]While floating-point texture formats are supported, they are very limited on GeForce 6 videocard series, and there is no support for single-channel floating-point textures in OpenGL 2.0 at all.

using the `glFrustum` matrix. The eye-space distance z_e is transformed as follows:

$$\begin{aligned} z_c &= -\frac{z_e\,(f+n)}{f-n} - \frac{2nf}{f-n} \\ w_c &= -z_e \end{aligned} \tag{3.9}$$

where z_e is the linear eye-space distance, n is the distance of the near clipping plane, f is the distance of the far clipping plane, and z_c and w_c are the clip coordinates that are used to compute the window-space depth.

This transformation is followed by a transformation to normalized device coordinates and then to window coordinates:

$$\begin{aligned} z_d &= \frac{z_c}{w_c} \\ z_w &= \frac{z_d + 1}{2} \end{aligned} \tag{3.10}$$

The depth buffer stores the z_w values. The depth-based edge detectors apply the inverse transformation to determine z_e (linear eye-space distance) from z_w:

$$z_e = -\frac{2nf}{(2z_w - 1)\,(f-n) - (f+n)} \tag{3.11}$$

This transformation must be applied to each sampled depth value, which makes the depth-based edge detector slower compared to the normal-based one. The edge detector could be sped up by using the second mentioned option — rendering the linear eye-space distances into a single-channel floating-point texture — when OpenGL support for single-channel floating-point textures matures.

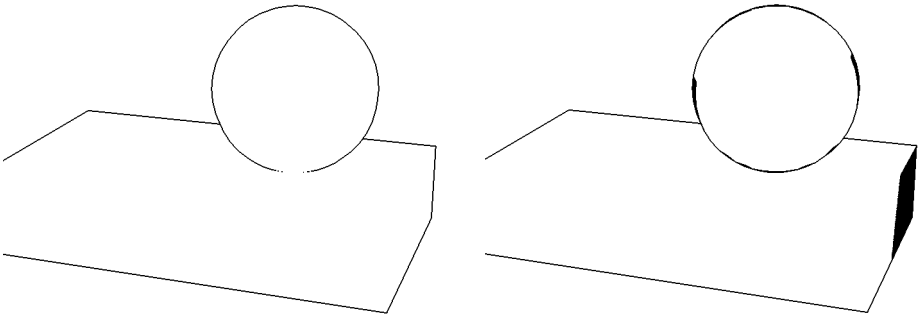
The first implemented edge detector uses a straightforward method to detect discontinuities in the depth buffer. It determines a maximum depth difference between each processed texel and its four neighbors:

$$d = \max\{|d_C - d_L|, |d_C - d_R|, |d_C - d_B|, |d_C - d_T|\} \tag{3.12}$$

where d_X denotes the linear eye-space distance at texel X. If d is greater than the specified threshold, the texel is classified as an edge. It is implemented as component `edge_detect_depth_deltas`.

This detector unfortunately requires precise fine-tuning of the threshold. When set too high, smaller discontinuities might not be detected. When set too low, polygons with high gradient might be falsely detected as edges. Both cases are shown in figure 3.11.

The second edge method detects discontinuities in the first derivative of depths, therefore it does not falsely classify polygons with a high gradient as edges. Moreover, it is also able to detect creases in the depth buffer. The component that implements this method is called `edge_detect_depth_2nd_derivative`.



(a) threshold = 0.3 (b) threshold = 0.05

Figure 3.11: Polygons with high gradient are falsely reported as edges when the threshold is too low.

The edge detector computes convolution of the depth image with a cross-shaped kernel that approximates the second derivatives, which produces the Laplacian of the depth image. Zero crossings in the Laplacian correspond to discontinuities in the first derivative of the depth image. Figure 3.12 shows the convolution kernel and the typical response of the applied Laplacian filter to a step edge. In this example, there is a discontinuity between pixels A and B. The implemented method classifies both pixels as an edge[3] by applying a threshold to the absolute value of the Laplacian. The absolute value at pixel C is computed as:

$$d = |d_L + d_R + d_B + d_T - 4d_C| \tag{3.13}$$

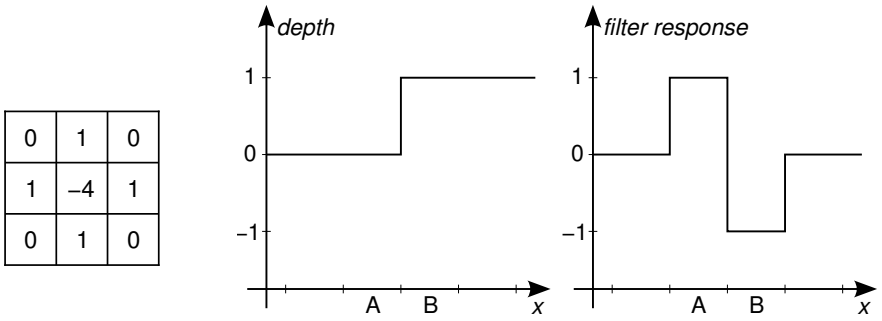
where d_X denotes the linear eye-space distance at pixel X, d is the absolute value of the Laplacian at the center pixel. If d is greater than the specified threshold, the pixel is classified as an edge. Figure 3.13 shows the method in more detail.

As with the normal-based edge detector, the thresholding can be optionally turned off. In that case the `edge_detect_depth_2nd_derivative` component displays the absolute value of the Laplacian multiplied by `threshold`.

It might seem the normal-based edge detector is not required — after all, silhouettes, border edges and creases can be detected in the Laplacian of the depth buffer. Unfortunately the depth-based edge detector has two major drawbacks:

1. Some creases have a small discontinuity in depth but a large discontinuity in normals. Pixels at such creases are correctly classified as edges by the normal-based edge detector. These creases can be missed in the Laplacian of depths because of the discrete sampling of the image, as shown in figure 3.14 on page 38.

[3] All implemented edge detectors search for edges at pixel boundaries, and classify pixels at the both sides of the edge as edge pixels. Therefore all detected edges are exactly two pixels wide. This usually produces more eye-pleasing results than single pixel wide edges.



(a) Convolution kernel (b) Response of 1-D filter to a step edge

Figure 3.12: Cross-shaped Laplacian convolution kernel, and response of its 1-D variant to a step edge.

2. Each polygonal mesh has a lot of creases — every edge shared by two polygons that are not coplanar could be classified as a crease. Several rendering methods exist to make a polygonal mesh look smooth. A common approach is to compute "smoothed" normals at polygon vertices, usually by averaging normals of all polygons sharing the vertex. Gouraud shading computes lighting at each polygon vertex using the smoothed normals, and then interpolates the computed vertex colors over the polygons. More precise methods, such as the Phong shading implemented by the `render_phong`, interpolate the vertex normals, and then compute illumination at each rasterized pixel in a fragment program.

   The input of the normal-based edge detector is rendered by the `render_normals` component, which draws interpolated normals, so the smoothing information is not lost. However, there is no such smoothing applied to the values stored in the depth buffer, therefore the depth-based edge detector has no way to distinguish "soft" and "hard" edges. Figure 3.15 on page 38 illustrates this problem.

This means that to detect silhouettes, border edges and creases correctly, both normal-based and depth-based edge detectors must be used. Figure 3.16 on page 39 shows an example where both detectors are necessary. Edges detected from normals are drawn as blue, edges detected from depth are drawn red, edges detected by both methods are black. The figure also demonstrates that "smooth" polygonal edges are falsely classified as creases when the depth threshold is set too low.

Surprisingly, the depth-based edge detector precision is good enough to detect polygon edges between nearly all polygons that are not coplanar, while not reporting false edges inside polygons, even when they are nearly perpendicular to the projection plane. Actually, all wireframe images in this thesis were generated by the depth-based edge detector.

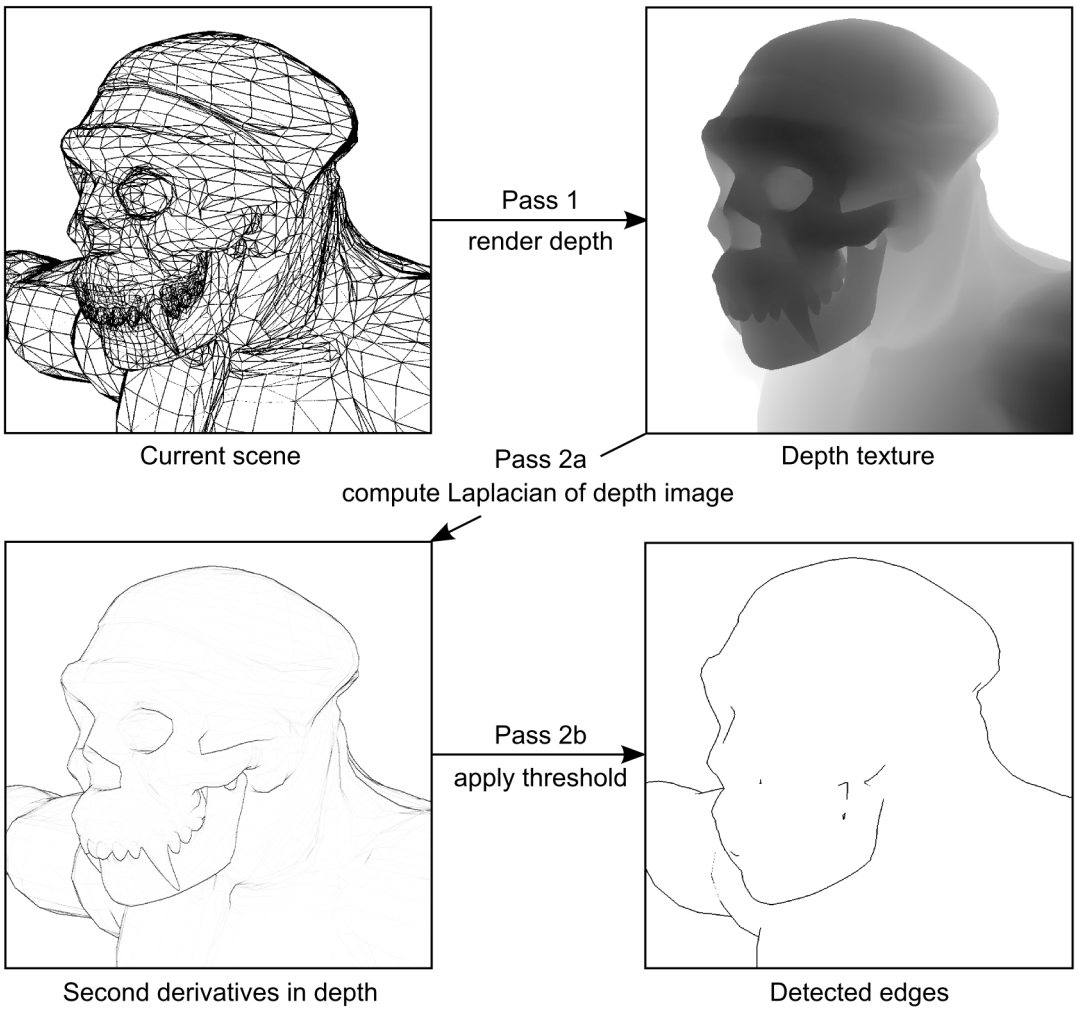


Figure 3.13: Depth-based edge detector.

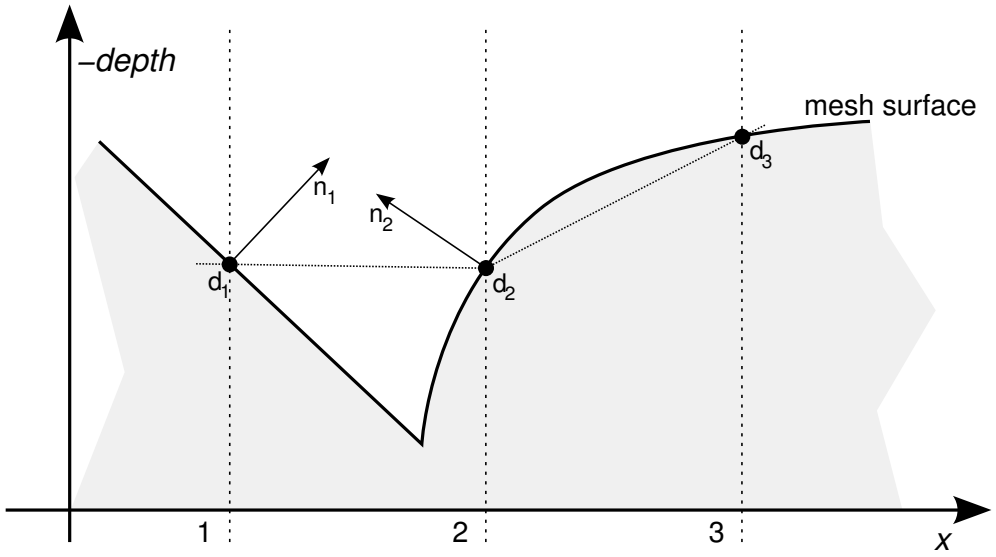


Figure 3.14: Ineffectiveness of the depth-based edge detector due to discrete sampling. The crease on the mesh surface is not detected by the depth-based edge detector, because there is no depth sample inside the valley. Normal-based edge detector is more effective in such cases.

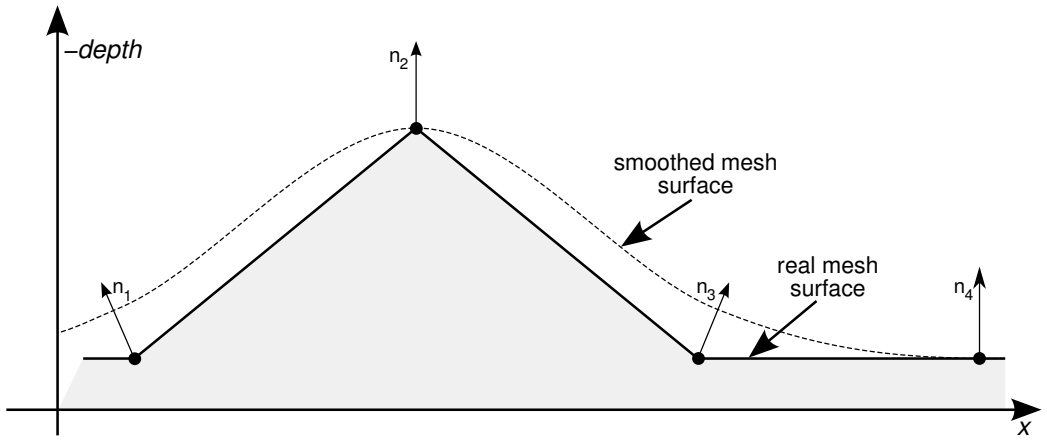
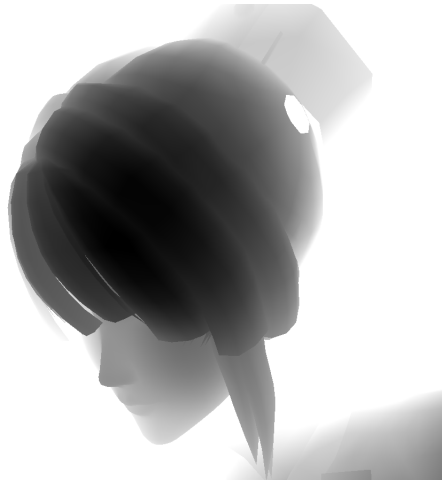


Figure 3.15: Smooth polygon edges incorrectly detected as creases. Because polygon smoothing is not reflected in per-pixel depths, edges that are treated as "smooth" by the normal-based edge detector may be classified as creases by the depth-based edge detector.



(a) Depths.

(b) Normals.

(c) Blue — edges detected from normals. Red — edges detected from depth.

(d) The same image with lower depth-based edge detector threshold.

Figure 3.16: Comparison of normal-based and depth-based edge detectors. The bottom-left image shows that both detectors must be combined to detect all silhouette and crease edges. The bottom-right image illustrates that the depth-based edge detector can not detect all creases. The hair creases are not detected even when the threshold is set low, because of the discrete sampling, while polygon edges that should be smooth start to show.

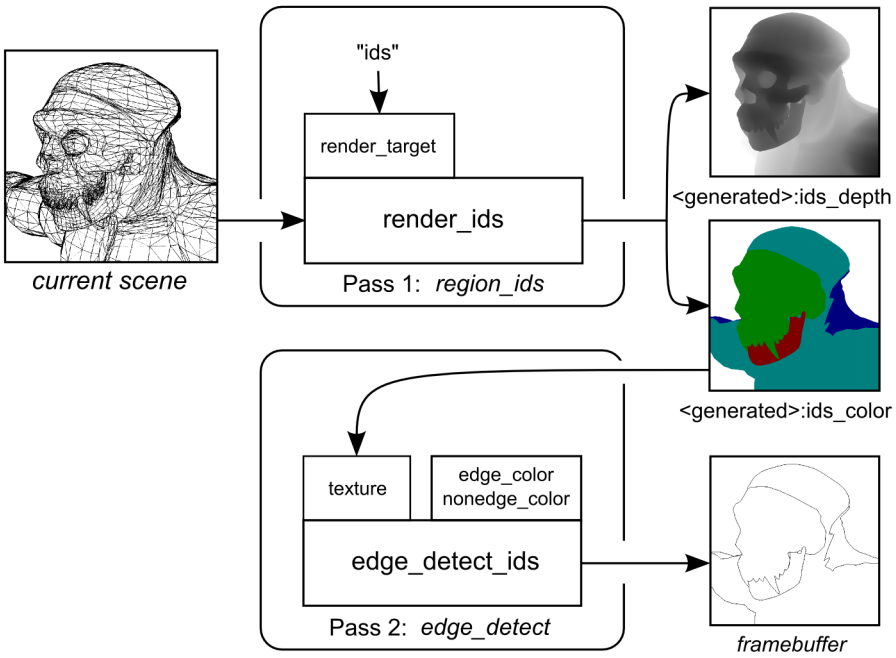


Figure 3.17: Detecting edges in region id texture.

3.3.3 Detecting edges in region identifier image

The only edge types that are not detected by the combination of the normal-based and depth-based method are boundaries between materials and artistic edges. Edges of this kind are detected by searching for discontinuities in a *region id* image. This approach is similar to that of Mitchell [25]. The region id image is constructed by component `render_ids`. It assigns a unique color to each mesh in the scene, and renders polygons of each mesh using this color. Component `edge_detect_id` then detects discontinuities in the region id image, as shown in figure 3.17. This approach heavily depends on proper separation of the scene into meshes.

3.3.4 Combined edge detector

Each of the three described edge detectors renders a different set of edges. The combined edge detector component `edge_detect` performs all three methods at the same time in a single pass. It requires three input textures — world-space normals, per-pixel depths and region identifiers — which can be generated in two passes. Therefore the complete edge detector technique consists of three passes, illustrated in figure 3.18. All other NPR techniques described in the following chapters use this three-pass edge detector.

At run-time, the component allows a user to tweak individual thresholds (or

switch thresholding off), to change output image colors, or to turn off the region id edge detector, in case the rendered scene is not properly partitioned into meshes as required by the detector.

Figures 3.19 and 3.20 show results generated by the combined edge detector. Edges drawn by the depth-based edge detector are shown as red, blue edges are detected from the normals and material boundaries are rendered as pale green. Edges detected by multiple methods are black.

3.4 Discussion and future work

The implemented algorithms are able outline the most important edges in polygonal meshes: silhouettes, border edges, creases and material boundaries. The edge detectors do not require any mesh preprocessing or run-time processing at all, they run completely on the GPU. To apply all three presented methods to a scene, only three rendering passes are required — the scene must be rendered twice to obtain all inputs for the combined edge detector, the detector itself consists of a single pass. The output is a single texture containing either a binary edge image with configurable colors (when thresholding is applied) or an image with edge intensities. Both outputs can be further processed, either to improve the edge look or to combine the results with other rendering techniques, as shown in the following chapters. However, no information about edge geometry is obtained, therefore the methods are not suitable for rendering artistic or suggestive edges, where object-space edge detectors are more suitable.

Compared to previously implemented methods, the use of 24-bit hardware depth buffer greatly improves the quality of the depth-based edge detector at a slight speed expense, especially when applied to more complex scenes, where 8-bit distances do not provide enough precision. The ability to tweak component variables (switching thresholding on and off, changing edge colors, modifying individual thresholds) at run-time also allows a user to obtain the best results for each scene and viewing angle.

The implementation could be improved to better utilize the most recent graphics hardware:

- Instead of using the depth buffer, another pass that would render linear eye-space distances to a single-channel floating-point textures could be used. This would eliminate the need to transform non-linear distances stored in the depth buffer in the edge detector fragment program at the expense of one extra pass, which could improve the processing speed.

- Utilizing the `ARB_draw_buffers` extension, which allows a fragment program to render to multiple color textures, the normal and region id images could be rendered in a single pass, further speeding up the technique.

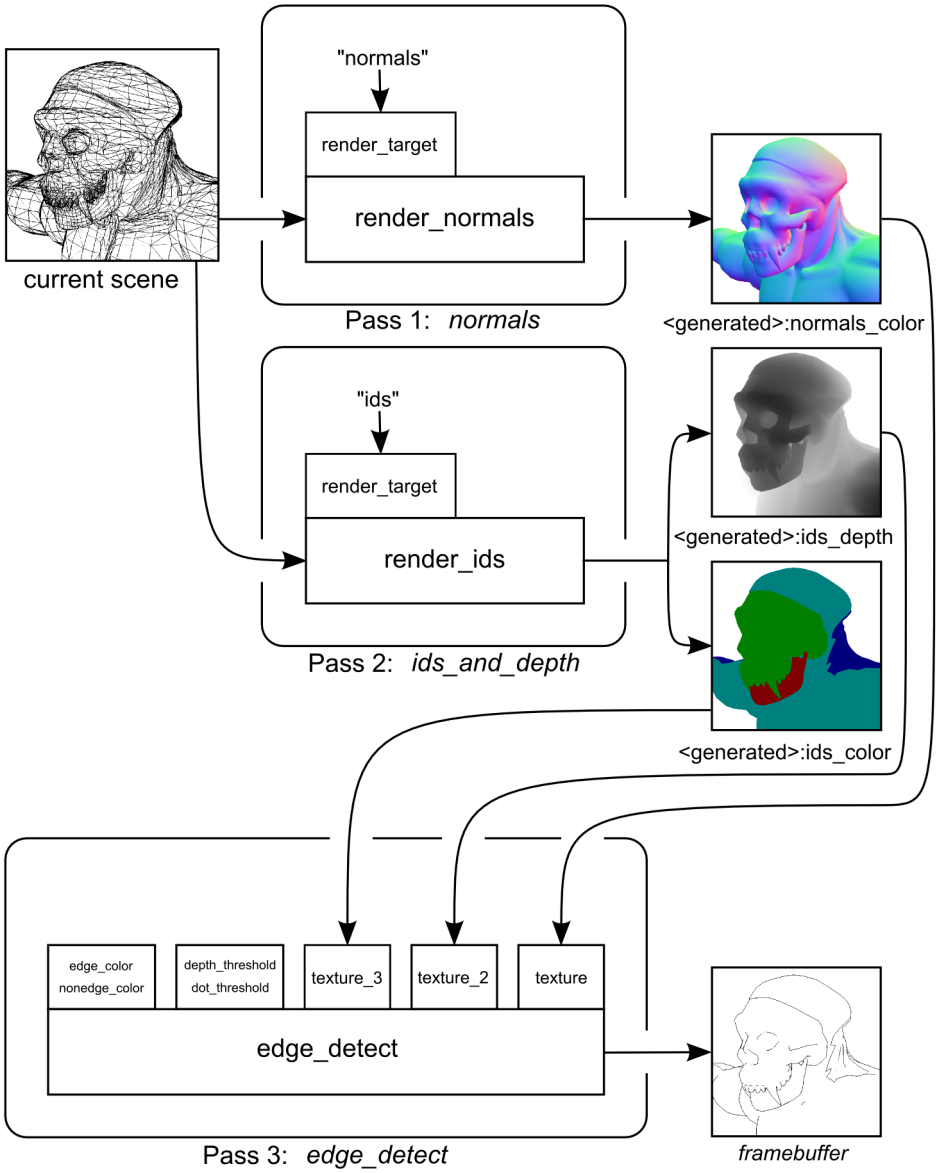


Figure 3.18: Combined edge detector.

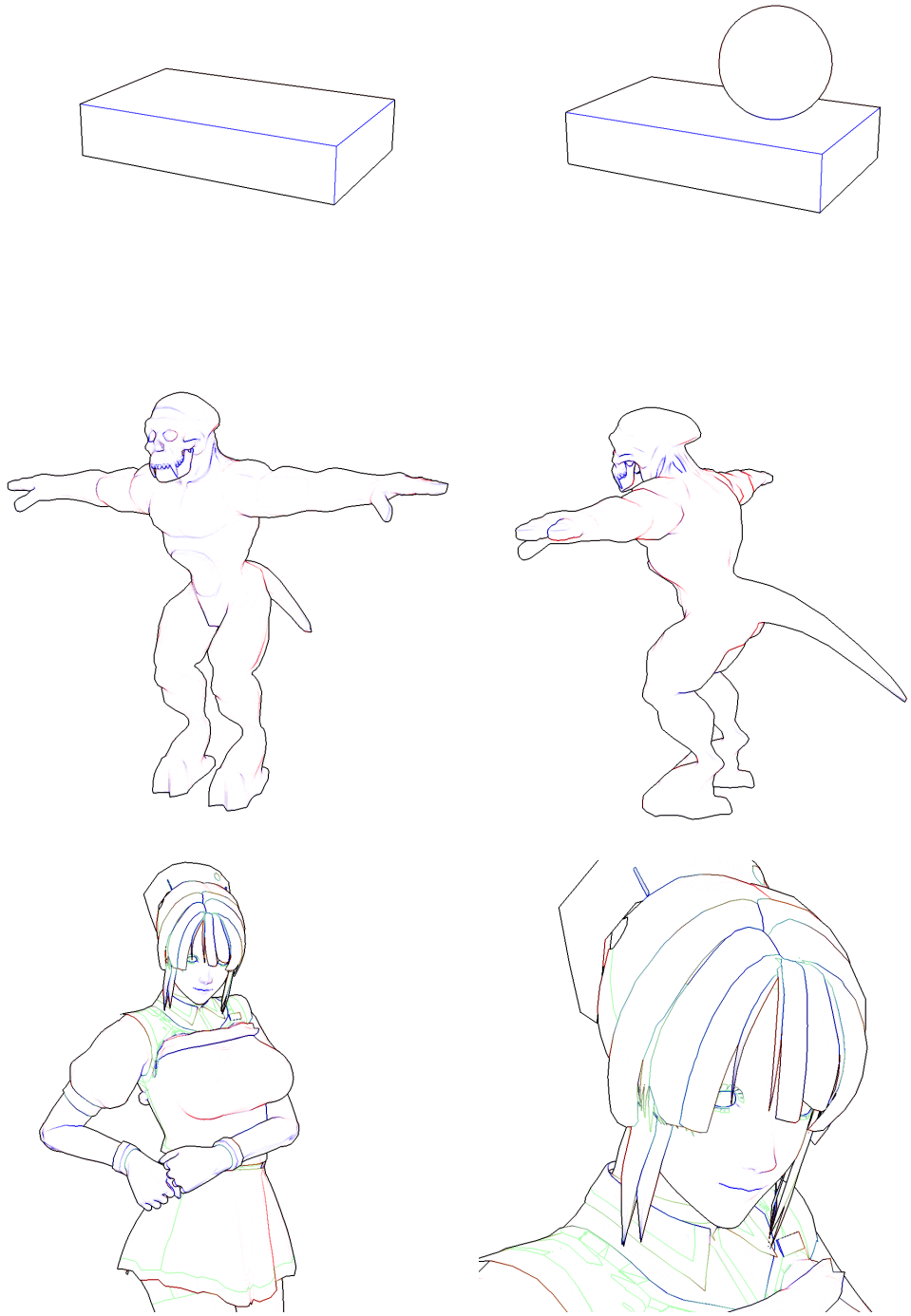


Figure 3.19: Images rendered by the combined edge detector.

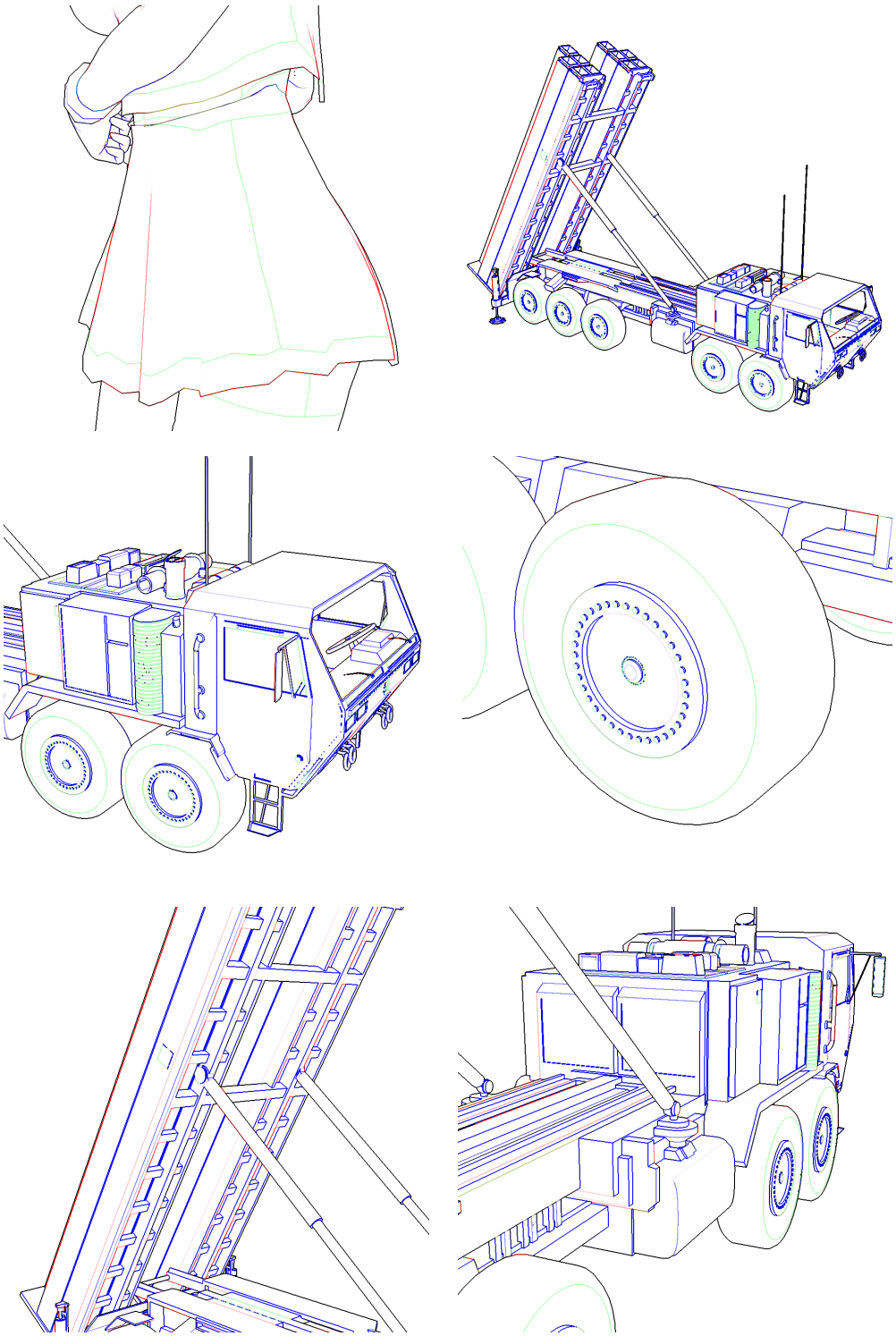


Figure 3.20: More images rendered by the combined edge detector.

Chapter 4

Cartoon-style rendering

Cartoon rendering, also known as cel-shading, combines two rendering methods: drawing object outlines (inking) and coloring cartoon objects (painting). Inking of 3-D models can be achieved by edge rendering methods discussed in the previous chapter.

When painting a cartoon character, the artist does not aim for three-dimensional appearance. Rather than trying to mimic realistic object shading, the artist uses low amount of solid colors to represent each object material. Some cartoons use only a single solid color, but usually two colors with the same hue and saturation are used to distinguish well-lit and shadowed areas of the object. Sometimes a third brightest area is used to for specular highlights. Boundaries between shadowed and illuminated colors are hard edges, therefore this painting style is sometimes referred to as "hard shading" or "stepped shading".

4.1 Previous work

Gooch et al. [10] proposed various rendering styles for technical illustrations. Although they do not implement the stepped shading, the presented variations of the standard lighting equations that achieve a warm-to-cool look suitable for technical illustrations are in many ways similar to the variations used in cartoon shaders.

Lake et al. [21] implemented the cartoon shading by computing a modified lighting equation at each vertex, and using the results as texturing coordinates to apply a one-dimensional texture called *toon map* to the object. When the texture contains only two colors and is rendered without filtering, a stepped cartoon look is achieved. Lake's method produces jagged artifacts when a transition between two color steps is observed at a close range at large polygons. Lake proposes to use a bilinear filtering when fetching the stepped intensity from the toon map, but then the transition would become smooth and could be too wide on large polygons.

This method was improved by Ishaya [16], who interpolates normals instead of intensities, and computes the lighting equation at each pixel using DirectX Pixel

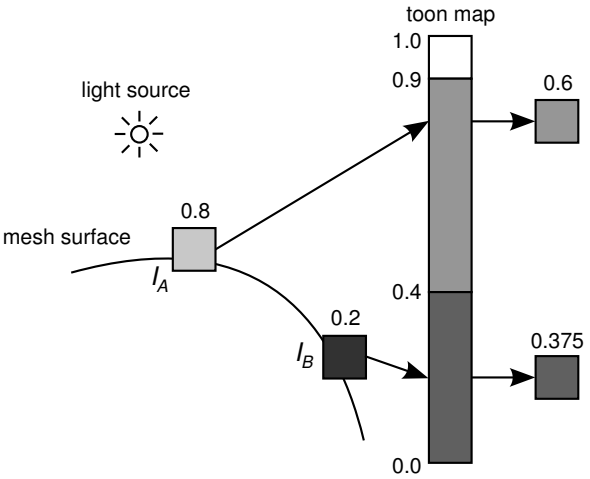


Figure 4.1: A toon map is used to transform smooth pixel intensities to stepped intensities.

Shaders 1.1. This approach also eliminates the artifacts.

Both Lake and Ishaya use only diffuse lighting model, therefore their algorithms do not display specular highlights.

A different approach that runs well on a non-programmable graphics hardware without any extra CPU processing was proposed by Dietrich [6]. First, the object is rendered flat-shaded using the bright color. In the next pass, the object is rendered with lighting turned on and an alpha test is used to discard all lit pixels, thus only the shadowed pixels are overlaid over the flat-shaded bright image. Claes et al.[2] process the geometry on the CPU to determine which triangles can be rendered simply as flat-shaded and subdivide the remaining ones in real-time.

4.2 Cartoon shader

The two cartoon shaders implemented in NPRView are based on techniques proposed by Lake et al. [21] and Ishaya [16]. It improves them by using a modified Phong shading model, thus adding realistic specular highlights to the rendered images.

The algorithms share the same basic idea. First, the *lighting intensity* is computed for the rendered pixel. The lighting intensity is a single scalar value that is used as a texturing coordinate into a toon map, which is a one-dimensional texture that usually contains only one to three colors. The texture lookup transforms smooth lighting intensities to stepped shading, as illustrated in figure 4.1.

Before describing the implementation details and comparing the differences between the two methods, let's examine the way the lighting intensity is computed. Each NPRView scene consists of one or more meshes and each mesh has a set of

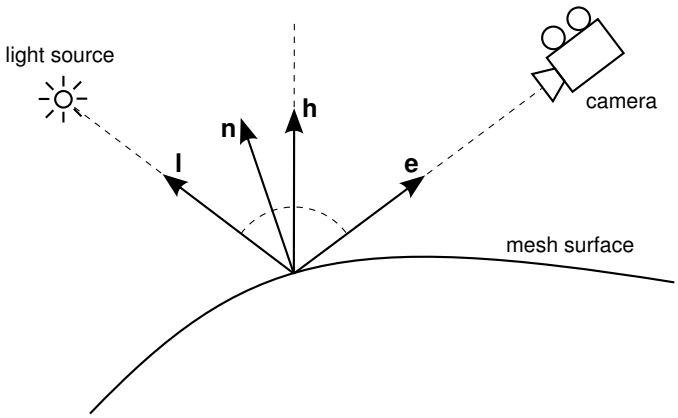


Figure 4.2: Phong illumination model. **n**: surface normal; **e**: pixel-to-eye vector; **l**: pixel-to-light vector; **h**: half-vector.

material properties. Since the scene is read from standard file formats (either OBJ or VRML), the material properties are suitable for the Phong illumination model, and must be somehow mapped to cartoon shader parameters.

A material has the following attributes: ambient color, diffuse color, specular color and shininess. When a mesh is lit by a single point light with no distance falloff, the following equations are used to compute the color at each lit pixel when the Phong illumination model is used:

$$C = a_m + c_l \left(f_d d_m + f_s s_m \right) \tag{4.1}$$

Here C is the resulting color, a_m, d_m and s_m are the ambient, diffuse and specular material colors, respectively, c_l is the light color, f_d and f_s are the diffuse and specular lighting factors computed as follows:

$$f_d = \max\{\mathbf{n} \cdot \mathbf{l}, 0\} \tag{4.2}$$

$$f_s = \begin{cases} 0 & \text{if } \mathbf{n} \cdot \mathbf{l} < 0 \text{ or } \mathbf{n} \cdot \mathbf{h} < 0 \\ (\mathbf{n} \cdot \mathbf{h})^s & \text{otherwise} \end{cases} \tag{4.3}$$

where s is the material shininess, **n** is the surface normal at the rendered pixel, **l** is the unit vector pointing from the lit pixel to the light source and **h** is the *half-vector*, computed as the normalized average of the pixel-to-light unit vector **l** and the pixel-to-eye unit vector **e**, as illustrated in figure 4.2.

The equation that computes the lighting intensity that is used to apply the cartoon shader in NPRView tries to retain all features of the Phong lighting model, namely the diffuse and specular components:

$$I = f_d + f_s \cdot \text{intensity}\left(s_m\right) \tag{4.4}$$

where I is the lighting intensity, f_d and f_s are the diffuse and specular lighting factors computed as specified in equations 4.2 and 4.3 and intensity (s_m) is the intensity of the specular material color, computed as:

$$\text{intensity}\,(c) = 0.3c_{red} + 0.6c_{green} + 0.1c_{blue} \tag{4.5}$$

Methods proposed by Lake [21] and Ishaya [16] use only the diffuse component f_d, therefore they do not display specular highlights.

After the value of pixel's I is determined, it is used as an index into a the toon map, which encodes the stepped cartoon intensity function, as illustrated in figure 4.1. The toon map is implemented as a one-dimensional texture, and I is used as a texturing coordinate to lookup the stepped lighting value, I_s. Then the following equation is used to determine the final color:

$$C_s = a_m + c_l I_s d_m \tag{4.6}$$

where C_s is the final pixel color, a_m is the ambient material color, c_l is the light color and d_m is the diffuse material color.

The difference between the two implemented methods is how the value of I is determined for a pixel. The first one, implemented by the `render_toon_per_vertex` component, computes the intensity values at each vertex, and interpolates them over the polygon. Unlike in the Lake's implementation, no artifacts at the step shading edges were observed, possibly due to the higher precision of the interpolators on the modern GPUs.

The second method, implemented by the `render_toon` component, interpolates the vertex normals over the polygons, and then computes the value of I at each rendered pixel in a fragment program.

The cartoon shading techniques that use these components further combine the images with output from the combined edge detector described in the previous chapter.

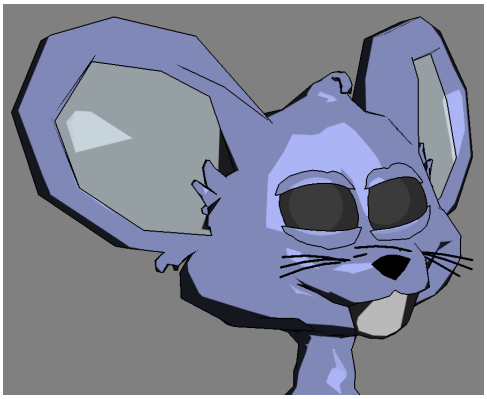
The first method has drawbacks similar to the Gouraud shading. On large polygons, the differences between interpolated colors (Gouraud shading) and colors determined by computing the illumination equation at each pixel (Phong shading) may become large, especially when the light is near the surface or when the material has a high specularity. Similar artifacts appear when the stepped shading model is used, as illustrated in figure 4.3. The quality of specular highlights is much better when the texturing coordinates into the toon map are computed per-pixel. In picture 4.3(c), the highlights even completely disappear.

To generate pictures 4.3(e) and 4.3(f), a special toon map containing the color spectrum was used. The toon map can be treated as a generic 1D function that maps intensities computed by the traditional illumination model to intensities of the rendered pixels. By using different toon maps, various appearances can be achieved as illustrated in figures 4.4 and 4.5. For example, one of the helmet pictures uses a gradient toon map that simulates a metallic look.

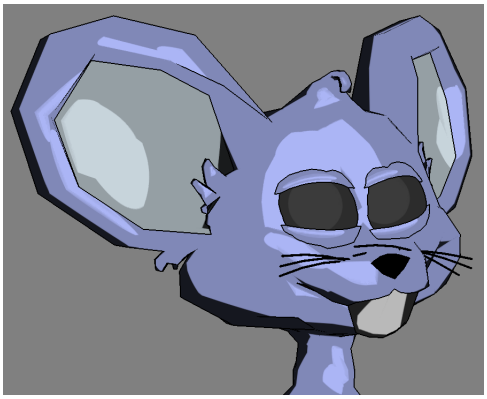
4.3 Discussion and future work

The implemented cartoon shader builds on works done by Lake et al. [21] and Ishaya [16] and further improves them. By using more modern GPU programming models with better precision, artifacts at boundaries between step colors were eliminated. Moreover, the shader supports specular highlights. When non-stepped toon maps are used, the scene can be rendered in different non-cartoony styles.

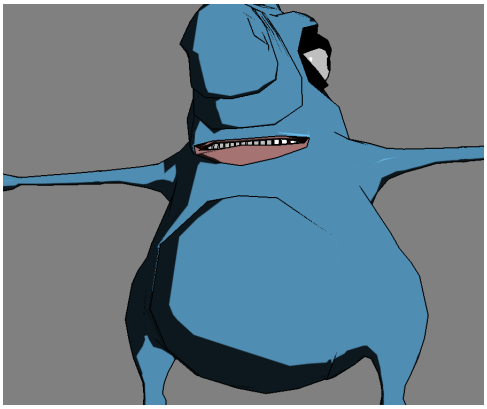
As a future work, the cartoon rendering technique could be extended to draw more stylized specular highlights, while still achieving real-time performance.



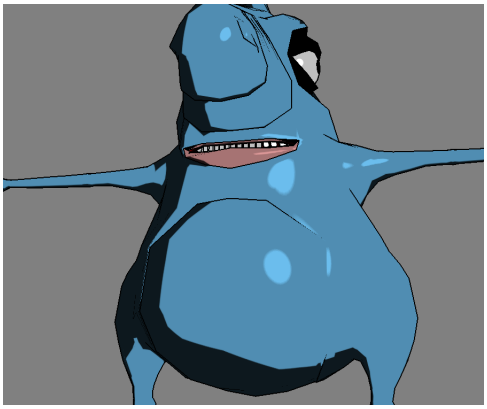
(a) Per-vertex, stepped toon map.



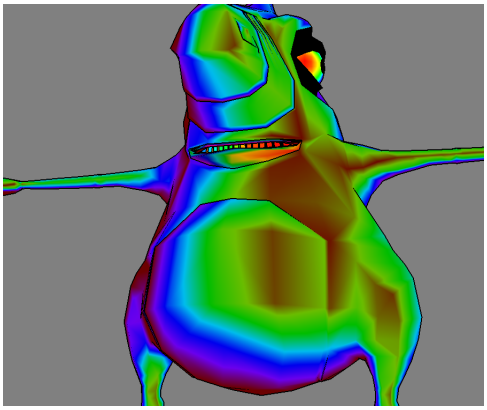
(b) Per-pixel, stepped toon map.



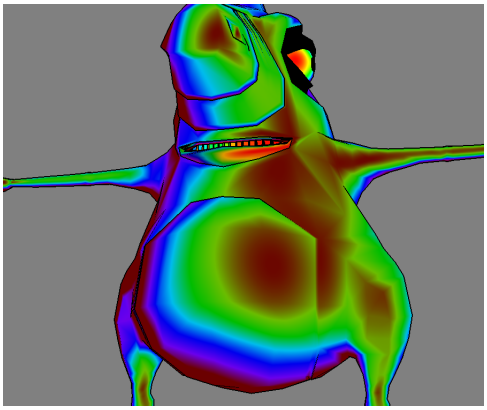
(c) Per-vertex, stepped toon map.



(d) Per-pixel, stepped toon map.



(e) Per-vertex, spectrum map.



(f) Per-pixel, spectrum map.

Figure 4.3: Comparison of `render_toon` and `render_toon_per_vertex`. On the left, output from method that computes toon texturing coordinates for each vertex and then interpolates them over the polygon is shown. On the right, vertex normals are interpolated over polygons and toon texturing coordinates are computed per-pixel.

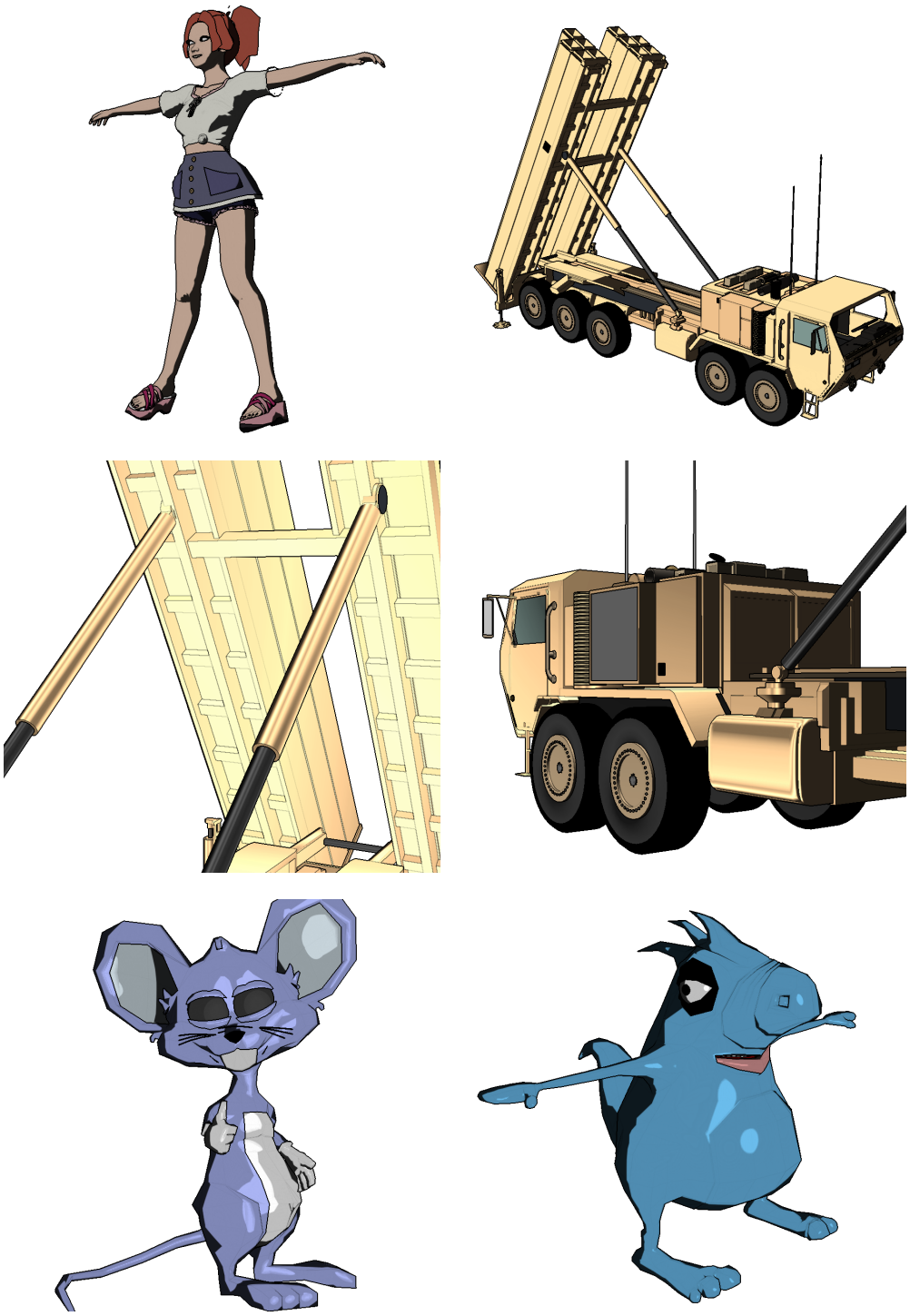


Figure 4.4: Images rendered by the cartoon shader.

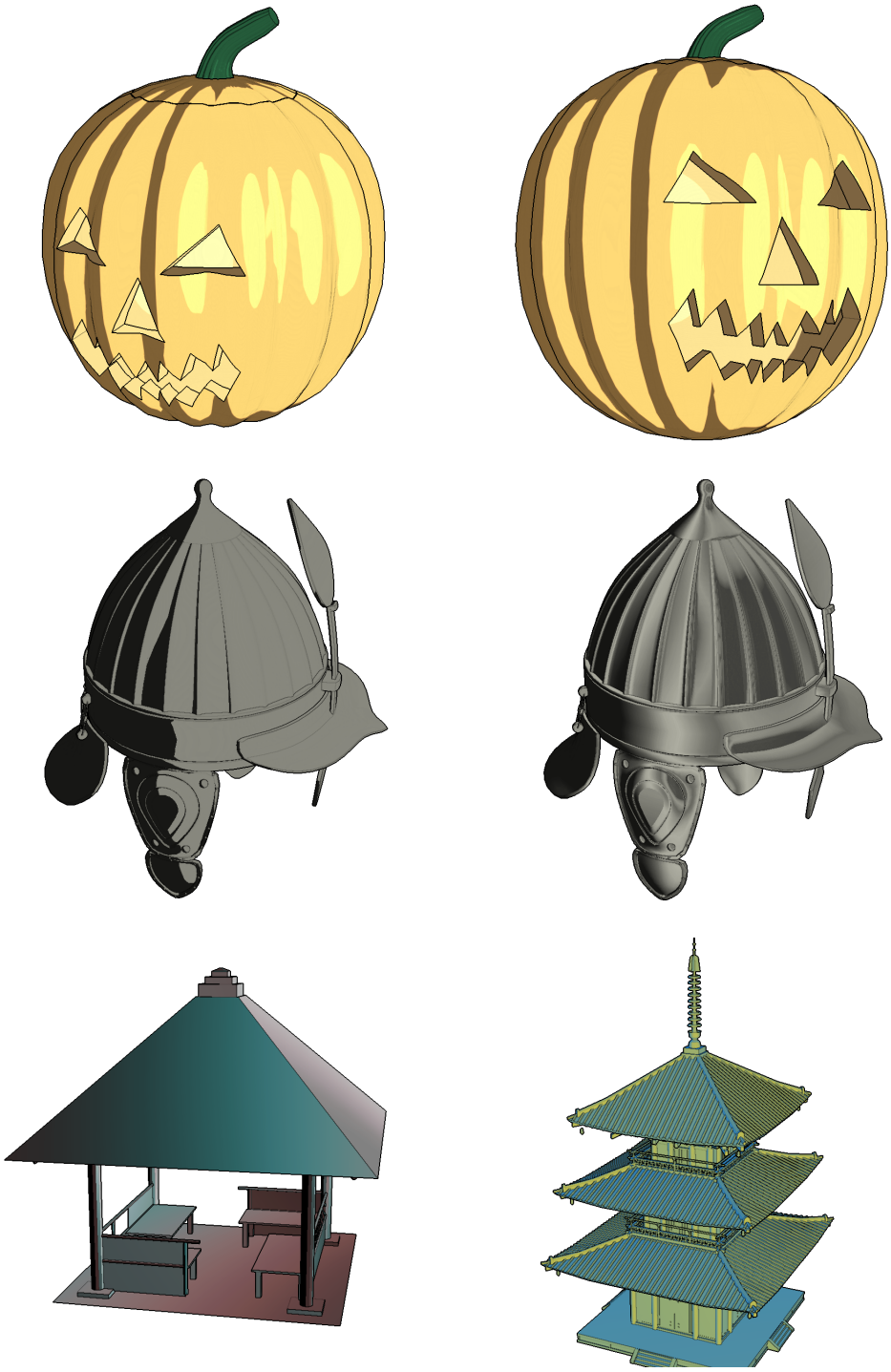


Figure 4.5: More images rendered by the cartoon shader.

Chapter 5

Mosaics

Mosaic is the art of decoration with small pieces of colored glass, stone or other material. Small tiles or fragments of pottery (known as tesserae) or of colored glass or clear glass backed with metal foils, are used to create a pattern or a picture. Figure 5.1 shows two mosaic examples.

(a) 'Cave Canem'. Pompeii.

(b) Mosaic at St. Monica Catholic Church in Mobile, Alabama, USA.

Figure 5.1: Mosaic examples.

Several rendering methods that replicate the look of classical mosaics were presented in the past. The non-photorealistic rendering proposed in this chapter tries to achieve similar results while rendering a generic three-dimensional scene at interactive framerates.

5.1 Previous work

After several years of mosaic rendering research, these observations regarding eye-pleasing packing of mosaic tiles were made:

1. Tiles should be evenly distributed; the amount of gaps and overlapping tiles should be minimal.

2. The orientation of tiles should emphasize edges in the source image. Tiles should not cross the edges.

3. Tile colors should approximate colors in the equivalent area in the same image. Both sampling the source color at the tile's centroid and computing the average of the colors in the tile's area give reasonable results.

Several works that try to effectively and efficiently solve the problem of packing tiles of arbitrary shapes and sizes into a given shape exist. Kim and Pellacini [20] proposed an algorithm that efficiently selects tiles from a large library in order to pack a 2-D container. For each tile configuration, energy E is defined as the sum of terms that penalize various negative aspects of the packing, such as mismatching color, large gaps between tiles or big tile overlap. The described algorithm searches for a tile configuration with minimal energy E.

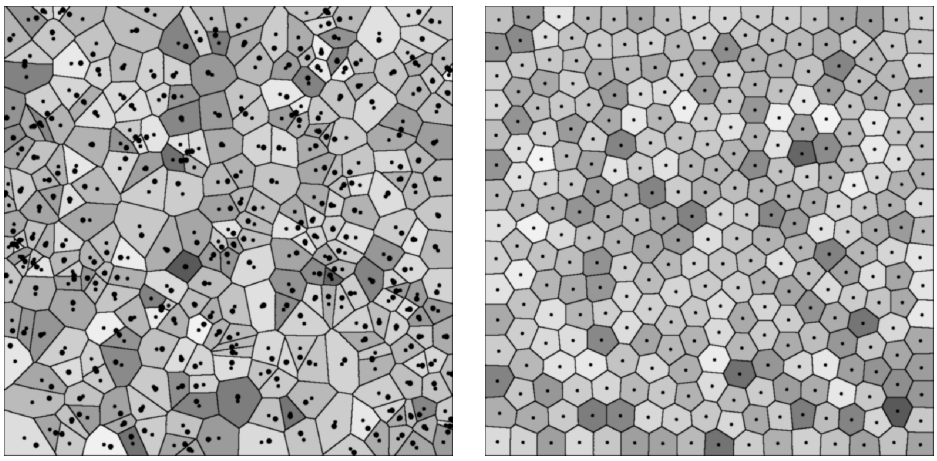
Recently, Dalal et al. [3] presented a similar technique that uses a metric that penalizes an overlap and gaps between tiles. The proposed algorithm uses the FFT in a novel way to simultaneously evaluate the metric for all possible pixel shifts of a tile.

Those methods achieve impressive results, but unfortunately they are computationally expensive, and not suitable for our primary goal — utilizing modern GPUs to their maximum to implement non-photorealistic rendering styles at interactive framerates.

To achieve a uniform spacing between mosaic tiles, Hausner [13]) and Smith et al. [31] use centroidal Voronoi diagrams (CVDs) and centroidal area Voronoi diagrams (CAVDs), respectively. These methods can fail to create even mosaic packing in some cases, but they're simpler and computationally faster.

A two-dimensional Voronoi diagram is defined by N points, called *sites*. The diagram partitions the plane into N regions, such that all points within a region are closest to its associated site. CVDs have one additional property — each site is located at the centroid of its region. Figure 5.2 shows an example of a Voronoi diagram and a CVD. CAVDs are similar to CVDs, but the sites can have any shape, therefore they are more suitable for packing generic (non-square) tiles into a mosaic.

As can be seen in image 5.2(b), CVDs produce hexagonal regions. To pack square tiles, a CVD that produces square Voronoi regions is required. Hausner achieves this by replacing the standard Euclidean distance by the Manhattan distance to determine which points are the closest ones to each site.



(a) Voronoi diagram with generators (large dots) and centroids (small dots).

(b) Centroidal Voronoi diagram.

Figure 5.2: Voronoi diagram.

To generate a Voronoi diagram, Hausner uses a method presented by Hoff et al. [19], first proposed by Haeberli [11]. The method takes advantage of the hardware depth-buffer. For each site point, an "infinite" pyramid with a unique color is rendered in orthogonal projection. Each pyramid's apex is positioned at the site point, all apices have the same depth-coordinate. The difference between a pyramid's apex depth and the depth of any point on the pyramid's surface equals to the Manhattan distance between the corresponding pixels in the image. Therefore if the pyramids are rendered with enabled depth testing, the rendered image is the Voronoi diagram for the Manhattan metric.[1]

Rong and Tan [28] presented a different algorithm to produce a Voronoi diagram, which is more suitable for implementation on GPUs. Hausner's approach requires drawing of a primitive of sufficient size (pyramid) for each site. On the other hand, the method by Rong and Tan generates a Voronoi diagram approximation in a constant time using a *jump flooding* algorithm. The technique requires only $O(\log n)$ passes over image of size $n * n$, independently on the number of used sites. The method is expected to produce good results with little errors event when applied to Voronoi diagrams with generalized site shapes — line segments, curves or areas.

CVDs can be easily produced using Lloyd's iterative algorithm [22]. At each iteration, each site is moved to its region's centroid, and the Voronoi diagram is recomputed. It's assumed this method converges in two dimensions, although the proof exists only for one dimension.

[1] Haeberli [11] draws cones instead of pyramids to compute Voronoi diagram for the Euclidean metric.

To compute tile orientations, Hausner uses a *direction field*, which specifies the orientation of the nearest edge for each pixel. This image can be constructed on the GPU. The algorithm requires edges defined manually by the user. First, an image which contains the distance to the nearest edge for each pixel is rendered utilizing the depth buffer in a way similar to Voronoi diagram construction presented above. The direction field is then computed as the gradient at each pixel of the distance image.

To ensure that the tiles do not cross the image edges, Hausner extends the Lloyd's CVD construction algorithm. When the edges are drawn as thick lines with a unique color over the Voronoi image, and several iterations of the algorithm are run, the centroids will be automatically displaced away from the edges. Then the lines are removed from the Voronoi image, and a few more iterations of the algorithm are run to fill the gaps.

The Hausner's approach has several drawbacks that limit its real-time implementation, as each iteration of the algorithm requires the CPU to read the contents of the framebuffer and process them in a non-trivial way. Also, to achieve proper orientation and edge avoidance, the user must manually define the edges.

In the following sections, more optimal implementation of the Lloyd's algorithm that runs completely on the GPU is presented and it is shown that even such implementation is still not fast enough to achieve interactive performance. A completely different approach to tile positioning, based on physical simulation of a spring system, is then presented. Finally, the image-space edge detectors are used to achieve proper tile orientation and edge avoidance.

5.2 Mosaic renderer overview

The mosaic rendering technique implemented in NPRView shares many core ideas with the algorithms based on Voronoi diagrams, but the process, while achieving similar visual results, is in many ways different. It tries to position a user-specified number of evenly-sized square tiles on the screen according to the "eye-pleasing mosaic" guidelines outlined above. Figure 5.3 shows a simplified overview of the individual technique passes.

The passes shown in the figure can split into three major groups:

Tile positioning These passes compute uniformly spaced positions of the rendered tiles. The initial positions are independent of the rendered scene.

Tile orientation and edge avoidance Information about the edges in the image is used to determine how individual tiles should be oriented and moved to emphasize the edge directions. The results of tile movement caused by the edge avoidance algorithm are reflected in the tile positioning data structures. Tile positioning then readjusts all the tiles again to retain the uniform spacing.

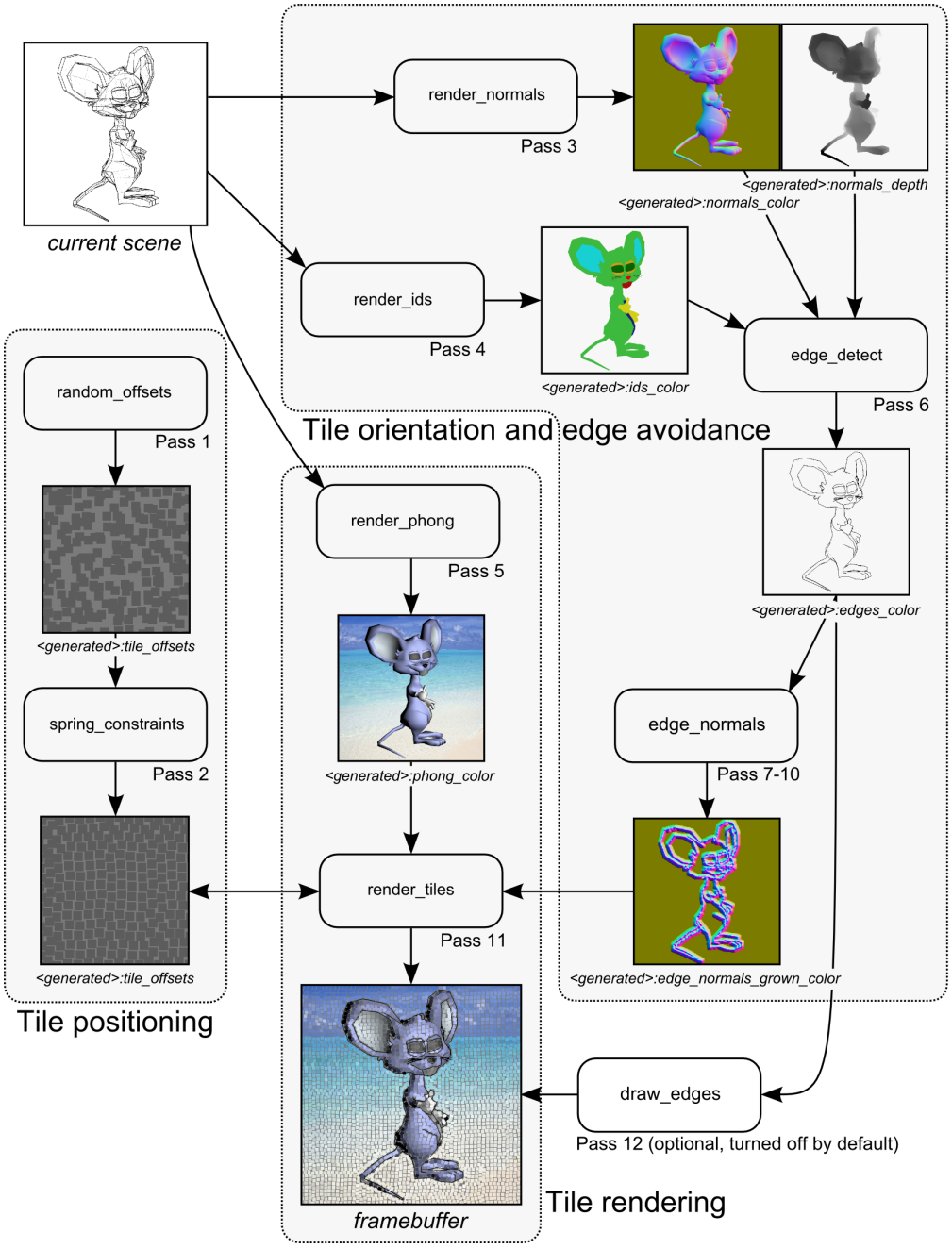


Figure 5.3: Mosaic rendering technique overview.

Tile rendering The data generated by the algorithms outlined above, along with a scene color rendering (using standard Phong shading, for example), are used to draw the tiles.

The following sections describe the individual pass groups in more detail.

5.3 Tile positioning

To position mosaic tiles, centroidal Voronoi diagrams are usually used. In this section, a GPU implementation of the Lloyd's algorithm is proposed. Unfortunately the implementation is not fast enough for visualization in real-time. The second part of this section presents alternative approach to the uniform tile spacing, based on the simulation of the system of infinitely stiff springs.

5.3.1 Constructing CVDs on GPU

Many fast algorithms that construct very precise approximation of a Voronoi diagram in two dimensions exist. However, no fast GPU implementation of the second step of the Lloyd's algorithm — computing the centroids of all Voronoi regions — was proposed yet. Hausner [13] reads the generated Voronoi image back from the framebuffer to compute the centroid positions on the CPU. While reading data from the video memory of a modern PCI Express graphics card is not as slow as it used to be a few years ago with AGP cards, the Lloyd's algorithm requires several iterations per frame. This is not an acceptable approach if we want to achieve interactive framerates.

The reason why such algorithm is hard to implement on a GPU is the *gather* nature of the fragment processing, as described by Harris in GPU Gems 2 [12]. To simplify a bit, a fragment program can easily read (gather) information from other parts of the processed texture, other textures available to the program and constant values uploaded into the GPU, but it's not capable of *scatter* — random-access writes. For example, there is no such feature as writable global variables available on GPUs. A fragment program can output one or more colors for the processed fragment and the fragment's depth, which gives us up to 17 floating-point scalar values, associated with the processed fragment.

Several methods of using the gather capability of fragment programs were researched during the implementation of the mosaic renderer. The most promising one was based on the idea of a *parallel reduction*. Reduction is an iterative process that shrinks the processed texture by some fraction in each step. For example, to compute a maximum value in a texture, each iteration computes a local maximum in a square region of four texels and writes a new texture of half size with the local maxima; each texel in the new texture is computed by gathering four values. After $O(\log n)$ iterations, a $1 * 1$ texture with the global maximum is obtained.

The reduction idea can be used to compute the centroid of a single Voronoi region. Let's assume we have a square area of width and height both equal to 2^n

for some n, and the Voronoi region is fully contained inside the area. Moreover, this area is available as a single texture. In the first step, the texture is processed and the following information about each texel is gathered:

count Number of texels in the area that belong to the Voronoi region. In the first step, this value is 1 if the texel belongs to the Voronoi region, 0 otherwise.

position_sum The sum of positions of all texels in the area that belong to the Voronoi region. In the first step, each texel is a single area by itself, so the *position_sum* is either the texel's position in the texture (if it belongs to the Voronoi region) or zero.

Then, $\log n$ reduction steps are applied to the texture, each halving its size. In each step a region of two-by-two texels is processed and the information about the region is computed:

count Number of texels in the $2 * 2$ area that belong to the Voronoi region. Computed as the sum of *count* of all four processed texels.

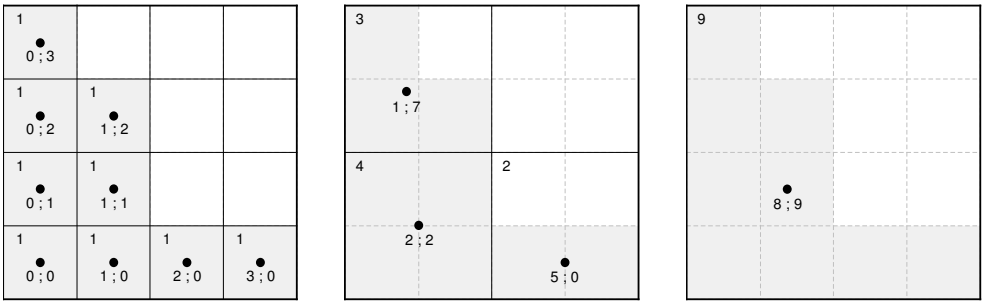
position_sum The sum of the positions of all texels in the area that belong to the Voronoi region. Computed as the sum of *position_sum* of all four processed texels.

This process is illustrated in figure 5.4. The result of the reduction is the number of texels belonging to the Voronoi region, and the sum of their positions. This allows us to compute the position of the centroid as:

$$\begin{aligned} c_x &= \frac{\sum_{t \in C} t_x}{|C|} \\ c_y &= \frac{\sum_{t \in C} t_y}{|C|} \end{aligned} \tag{5.1}$$

where and C is the set of all texels belonging to the same Voronoi region and $(c_x; c_y)$ is the position of the centroid of the Voronoi region.

This algorithm has many performance and implementation issues. As described here, the algorithm requires several rendering passes to compute the centroid of a single Voronoi region. If it was applied to each region separately, it would not be possible to achieve interactive framerates — $O(m \log n)$ passes would be needed to perform a single iteration of the Lloyd's algorithm, where m is the number of tiles and n is the size of the largest bounding area of a tile. The algorithm could be optimized to perform reduction for multiple Voronoi regions at the same time, as long as their bounding areas do not intersect. Another option would be to extract the bounding areas of all m tiles and pack them into a single texture. This can be done in a single rendering pass by drawing m screenspace-aligned quads. Then the centroids of all m tiles can be computed in parallel in $\log n$ reductions, albeit on a very large texture.



(a) Image before the reduction. (b) After the first reduction step. (c) The result after the second step.

Figure 5.4: Computing centroid of a Voronoi region. The dark pixels belong to the Voronoi region. For each square area, the values of *count* and *position_sum* are displayed, along with dots that denote centroid positions of the area.

Still, $\log n$ reductions per one Lloyd's iteration is too many. As Hausner [13] observed, approximately 20 iterations are required to compute satisfactory CVD, with additional iterations needed to apply the edge-avoidance technique. Therefore the proposed GPU implementation would require hundredths of passes. In the end, the reduction algorithm was not used in NPRView because of these issues.

5.3.2 Our approach

Instead a completely different approach to compute evenly-spaced tile positioning is used. When each tile is represented by a particle on a plane that is connected by a spring to its neighbors, the spring system ensures that the distances between the neighbor particles (tiles) are uniform. When the springs are infinitely stiff, the system can be efficiently solved on the GPU.

The algorithm is based on the cloth physics solver developed by Jakobsen [17]. He represents a cloth patch as a set of particles connected by infinitely stiff springs in a regular grid pattern. Such system can be easily solved by a constraint solver using relaxation. This method works by consecutively satisfying various local constraints and then repeating the process until the configuration converges to a global configuration that satisfies all the constraints at the same time.

First we examine how an infinitely spring constraint between two particles X and Y can be solved. The particles are connected by a stick (infinitely stiff spring) of length l. Assuming the current distance between the particles is d, the following equations determine the new positions of the particles (X' and Y') that satisfy

the distance constraint:

$$\begin{aligned} \mathbf{x} &= Y - X \\ d &= |\mathbf{x}| \\ \mathbf{u} &= \frac{d - l}{2d}\mathbf{x} \\ X' &= X + c\mathbf{u} \\ Y' &= Y - c\mathbf{u} \end{aligned} \tag{5.2}$$

where c is the *relaxation coefficient*. When c is equal to 1, the result satisfies the constraint after one iteration. Using under-relaxation ($c < 1$) or over-relaxation ($c > 1$) might lead to more stable results or a faster convergence when the system of such constraints is solved. Figure 5.5 illustrates this set of equations for $c = 1$.

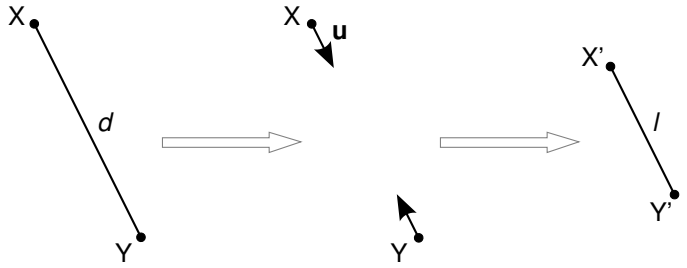
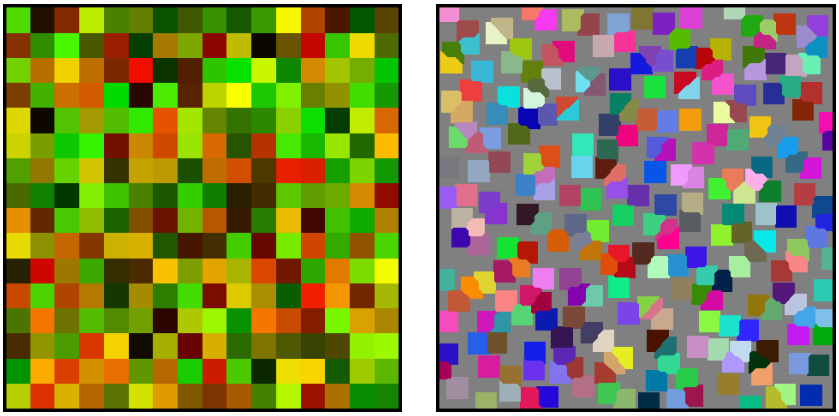


Figure 5.5: Solution of a simple system with two particles connected by an infinitely stiff spring.

In NPRView each tile is represented by a single particle. The particles are arranged in a regular grid, each is connected to its four neighbors by an infinitely stiff spring. When such spring constraint system is solved with under-relaxation, the particles will be nearly evenly spaced, while not positioned completely regularly, in a way similar to positions of centroids when a CVD is generated.

The particle positions are stored in a texture, each particle is represented by a single texel. Red and green channels of the texels contain particle offsets relative to the regular grid positions. First, the `generate_texture` component is used to generate the offset texture by assigning a random offset to each particle. The number of rendered tiles can be changed by modifying the resolution of the texture. Component `process_texture_spring_constraint` then applies the specified number of constraint solver iterations to the texture. These two passes are shown as pass 1 and pass 2 in figure 5.3 on page 57.

The initial random texture is generated only once when the technique is initialized, or when the parameters of the component (resolution, for example) are changed. The texture then remains persistent — in each frame, it is updated by the constraint solver and the tile renderer, as described in section 5.5.1 on page 67, but it is never randomized again. The persistence of the tile offset texture ensures



(a) Initial random offsets.

(b) Corresponding tile positions.

(c) Offsets after one iteration.

(d) Positions after one iteration.

(e) Offsets after five iterations.

(f) Positions after five iterations.

Figure 5.6: Tile positions computed by the spring system method.

that the tiles do not change their positions randomly or abruptly from frame to frame.

The constraint solver component computes new positions (X' in equation 5.2) of all particles at the same time in a single pass. It runs completely on the GPU: for each texel, offsets of its four neighbors are read from the offset texture, and the new offset of the particle corresponding to the texel is computed as specified in equation 5.2. The number of applied iterations is configurable by the `pass_count` component variable. The relaxation coefficient c and the spring rest length l are also exposed as component variables, and can be tweaked at run-time to produce the most eye-pleasing tile configuration.

Experiments show that approximately five iterations with relaxation coefficient 0.9 and rest length 1.005 generate a nice tight tile packing with little gaps or overlapping, while not packing the tiles in a regular grid, as shown in figure 5.6. This is a huge improvement over the hundredths of passes required to compute the CVD on the GPU.

5.4 Edge direction image

So far we managed to evenly distribute the tiles in a non-regular pattern, completely ignoring the scene that should be rendered. The main trait that makes mosaics visually appealing is the placement and orientation of the tiles that emphasizes edges in the source image. This section describes how an *edge direction image* is constructed. The image contains approximate information about orientation and distance of the nearest edge for all pixels within configurable distance of the image's edges, stored in the red, green and blue channels as follows:

- A unit vector pointing away from the nearest edge is encoded in the red and green channels. The orientation of tiles should be perpendicular to this vector. For simplicity, these vectors are referred to as edge normals.
- The *influence* of the nearest edge is stored in the blue channel. The influence decreases with the increasing distance of the pixel from the edge by a user-specified amount.

The edge direction image is used to orient tiles and to ensure they do not cross the image edges. The tiles are rotated according to the orientation of the nearest edge. The amount of rotation is controlled by the influence of the nearest edge — tiles adjacent to edges are oriented parallel to the edges, edges further away are axis-aligned. This tile orientation technique is used in mosaic shown in image 5.1(a) on page 53.

Tiles crossing the image edges are pushed away from the edges. This position change is reflected in the spring system structures, therefore the surrounding tiles will be automatically moved to retain uniform spacing by the spring system solver.

In NPRView no explicit edge definition is available to generate the edge direction image. The edge combined detector described in chapter 3 is applied to the

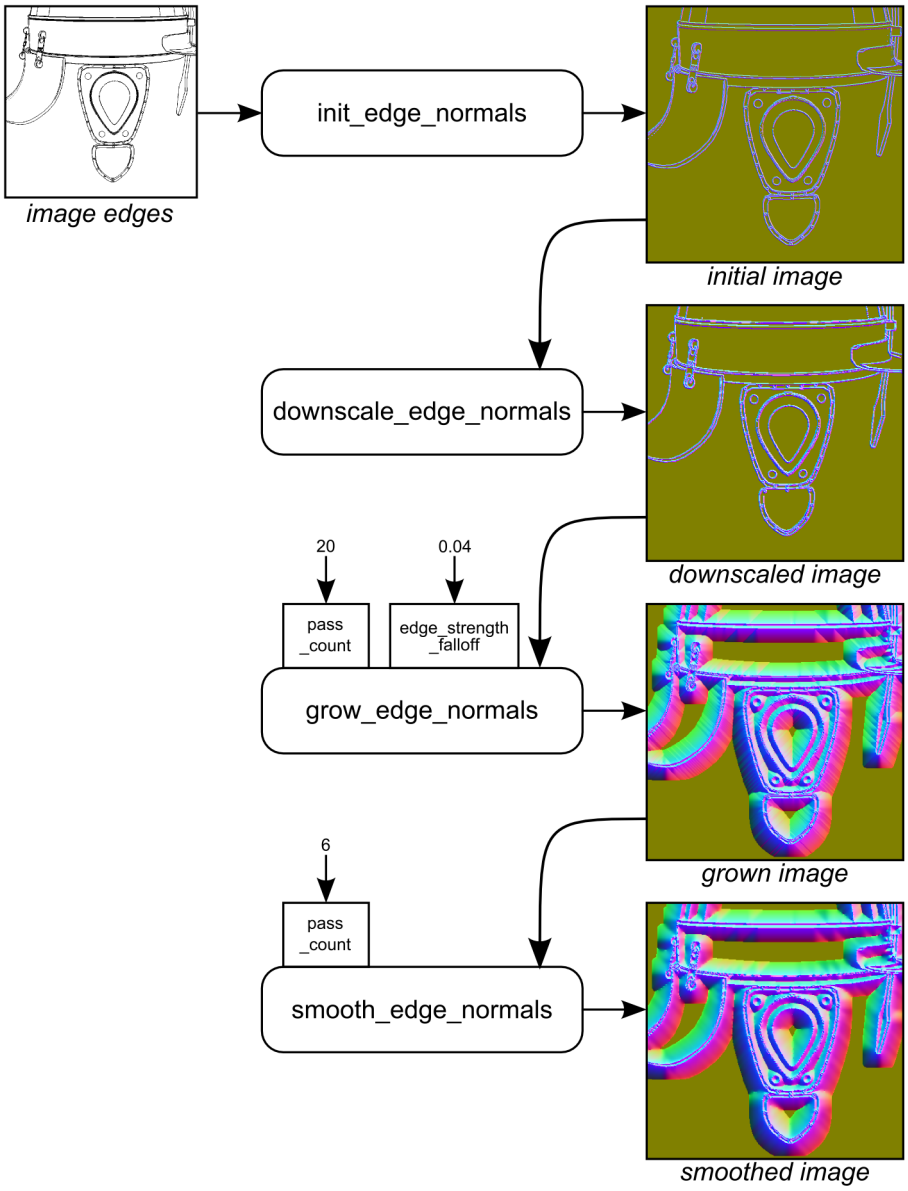
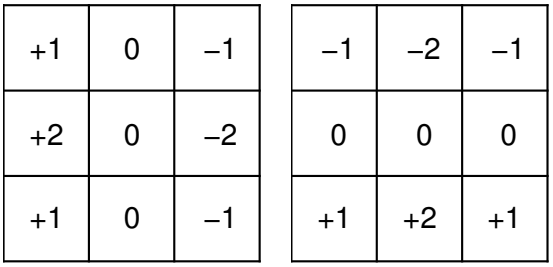
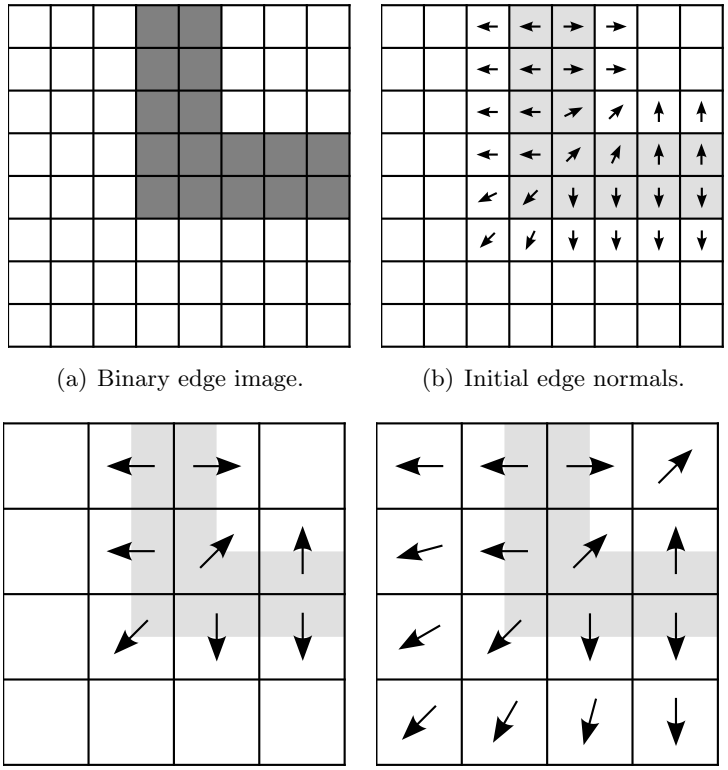


Figure 5.7: Generating edge direction image.



(a) G_x (b) G_y

Figure 5.8: Sobel operator convolution kernels.



(a) Binary edge image. (b) Initial edge normals.

(c) Downscaled edge normals. (d) Edge normals after one grow iteration.

Figure 5.9: First steps to generate edge normals.

scene to generate a binary image that classifies all image pixels as either edges or non-edges, as illustrated in figure 5.3 on page 57 in passes 3, 4 and 6. The edge image is then processed in four passes (labeled as 7–10 in the figure) to generate the edge direction image, as illustrated in figure 5.7. The four passes are:

Initialize edge direction image The binary edge image is first processed to generate initial edge normals at pixels adjacent to the edges. The Sobel operator is used to compute gradients G_x and G_y in the x and y axes, respectively. The operator is implemented as the convolution with two kernels (one for each axis) shown in figure 5.8. At pixels where the gradient is non-zero, the normal is computed by normalizing the vector $(G_x; G_y)$, a zero normal is assigned to the remaining pixels.

This pass is implemented by the `init_edge_normals` component. It reads the specified edge image and generates image with initial edge directions.

Downscale the image The edges generated by the algorithm described in chapter 3 are always two pixels wide and the output from the Sobel operator is non-zero at both the edge pixels and pixels directly adjacent to edges, as shown in figures 5.9(a) and 5.9(b). Therefore the regions with non-zero normals in the initial edge direction image are four pixels wide, with two pixels at each edge side, and the image can be downscaled to half resolution without any significant loss of information. This speeds up consequent edge direction image processing. An example of a downscaled edge direction image is shown in figure 5.9(c).

This pass is implemented by the `downscale_edge_normals` component. It reads the specified edge direction image and generates texture with half width and height containing the downscaled image.

Grow edge direction image In this pass, the information the about edge directions is distributed among other pixels in the image. Interpolating the normals across the whole image is not a simple task. Johnston [18] uses a system of dampened springs to compute the directions, but such system is very large compared to the tile positioning spring system used in NPRView and requires too many iterations to be properly solved.

Instead, we opt to determine the directions only within a certain vicinity of the edges. This is a reasonable limitation, since many real world mosaics are constructed by using only one or two rows of tiles to emphasize the edges, with the remaining space filled in a more regular pattern, as shown in image 5.1(a).

A form of dilatation is applied to the edge direction image to cover sufficiently large areas surrounding all edges. For each pixel with unassigned edge normal, the sum of the eight adjacent edge normals is computed. If the sum is non-zero, the result is normalized and the edge direction image is

updated for the processed pixel. Thus the covered area grows by one pixel in each iteration.

This pass is implemented by the `grow_edge_normals` component. It processes the specified edge direction image inplace. Component variable `pass_count` specifies the number of grow iterations, variable `edge_strength_falloff` controls the falloff of edge influence with distance from the edge. When set to $1/pass\_count$, the influence will drop exactly to zero at the most distance pixels. As the influence directly affects how much a tile's orientation should follow the edge direction, it is sometimes useful to set the variable to a lower number to rotate even the most distant tiles. When set to zero, all tiles in the area affected by edge direction image will be oriented parallel to the nearest edge.

Smooth edge direction image Because the input to this process is a discrete binary edge image, the edge normals determined in the first pass may contain abrupt changes from pixel to pixel. Some of these artifacts remain when the image is grown. To prevent irregular tile orientation, the resulting edge direction image is smoothed by averaging edge normals in a cross-shaped neighborhood of each smoothed pixel. Special care must be taken when smoothing the image; pixels adjacent to edges are not smoothed to prevent edge normal "bleeding" across edge boundaries.

This pass is implemented by the `smooth_edge_normals` component. It processes the specified edge direction image inplace. Component variable `pass_count` controls the number of smoothing iterations applied to the image.

5.5 Tile rendering

After both images that define the tile positioning (tile offsets and edge directions) are generated, the only other input required before the tiles can be drawn is a texture specifying the tile colors. In the implemented technique, the scene is rendered using the standard Phong shading, combined with a background image which greatly increases the visual appeal of the generated mosaic. However, a cartoon shaded image could be also used to generate a more "flat"-looking mosaic.

To draw the tiles, the tile offset and edge direction images are read from the video memory and processed on the CPU by the component `draw_tiles` to generate the tile geometry, as illustrated in figure 5.10. The following subsections describe the process.

5.5.1 Edge avoidance

The tile positions are read from the tile offset texture. However, these positions do not reflect the positions of the image edges and might cross them. To prevent

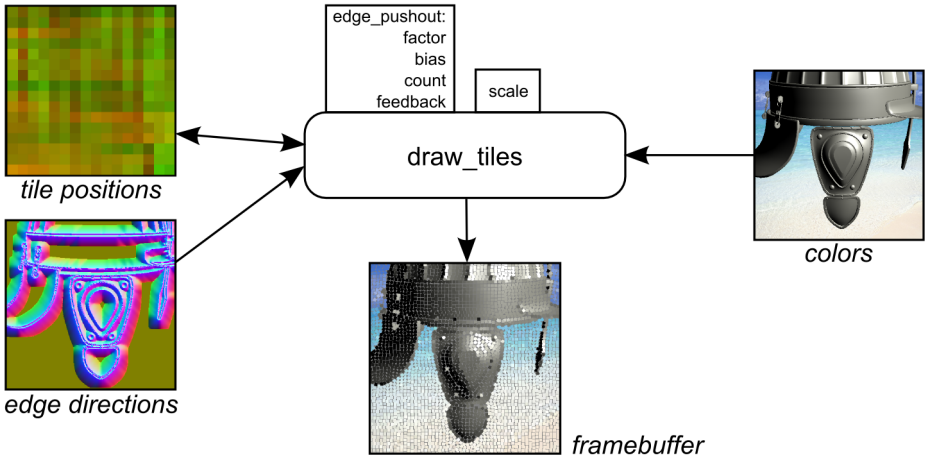


Figure 5.10: Component `draw_tiles`.

this, the tiles are moved according to information obtained from the edge direction image.

For each tile, the component `draw_tiles` applies `edge_pushout_count` iterations of "edge pushout". In each iteration, the tile is moved along the edge normal sampled from the edge direction image at the tile's position by a certain distance:

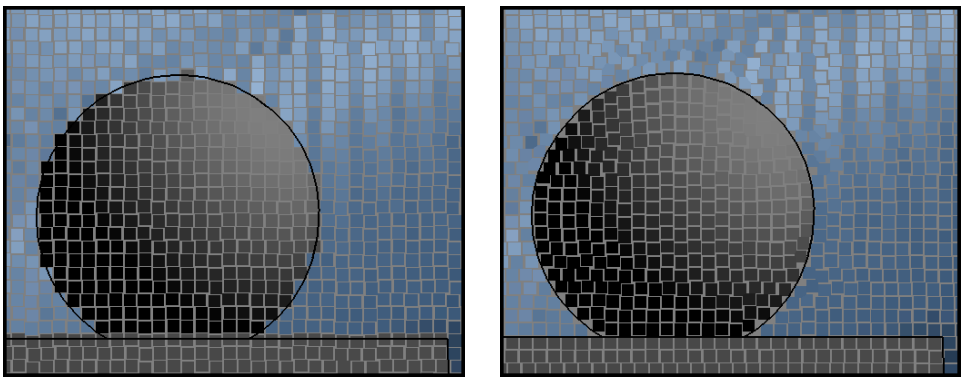
$$\mathbf{x}' = \mathbf{x} + d\mathbf{n} \tag{5.3}$$

$$d = \max\{f(i-b), 0\} \tag{5.4}$$

In this equation, $\mathbf{x}$ denotes the tile's position before the edge pushout is applied and $\mathbf{x}'$ the new tile position. The tile is moved along the edge normal $\mathbf{n}$ by d units, where one unit corresponds to the distance between two neighbor tiles if they were arranged in a regular grid. Value of d is computed as specified by equation 5.4, where i is the edge influence sampled from the edge direction image, b is the value of the component variable `edge_pushout_bias` and f is the value of the component variable `edge_pushout_factor`. The bias b specifies the lowest influence that still affects the tile's position and the factor f determines how large are the individual pushout steps. When the factor f is low, the tile position's converge slowly, when it is set too large, the pushout may become unstable.

After all tiles are moved, the position offset image is updated to reflect the new positions. The `edge_pushout_feedback` can be used to control the amount of feedback. The position offset of a tile is set to $p_{new} = fp_{pushed} + (1-f)p_{old}$, where p_{pushed} is the tile position after the pushout and p_{old} is the original tile position.

When configured properly, the pushout positions all tiles in the vicinity of an edge nicely along the edges. Because the pushout variables can be changed at run-time, the settings can be tweaked to produce the best results for given scene, viewing angle and configuration of the `grow_edge_normals` component.



(a) Original tile positions. (b) Positions after edge pushout.

Figure 5.11: Tile positions computed by the edge avoidance algorithm.

Figure 5.11 shows how the tile positions change when the edge pushout is applied.

5.5.2 Tile orientation

After the tiles are moved to positions that do not cross the image edges, they are rotated to further emphasize the edge directions. Information from the edge direction image is used again to achieve this. The edge normal specifies the tile orientation, the influence determines whether the tiles should be oriented parallel to the edge or whether the neutral axis-aligned orientation should be used. The orientation angle θ is computed as:

$$\theta = i\phi + (1 - i)\phi_0 \tag{5.5}$$

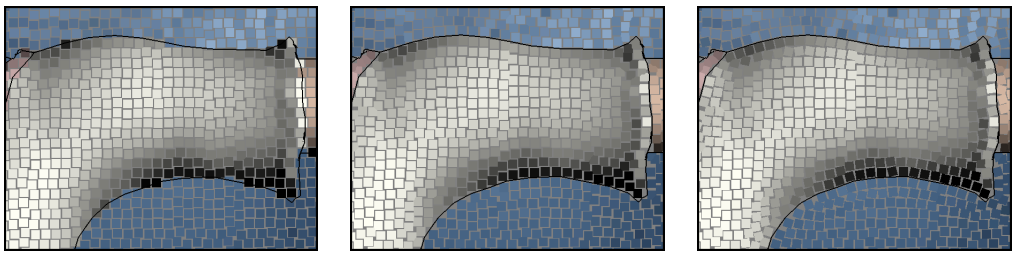
where ϕ is the orientation perpendicular to the edge normal, ϕ_0 the nearest axis-aligned orientation to ϕ and i is the edge influence.

Figure 5.12 illustrates the differences between the edge-independent tile positions, the positions after the edge avoidance and the final tile placement after tile orientation is changed to emphasize the image edges.

5.5.3 Drawing the tiles

Finally, each tile is rendered in orthogonal projection as a pyramid. By rendering pyramids instead of squares, the depth buffer can be used to produce a better looking results when two tiles overlap. All vertices of the pyramid have the same texturing coordinates that are used to sample the color image at the tile's center. Optionally, the edge image may be overlaid over the mosaic.

Figures 5.13 and 5.14 show mosaics generated by the proposed rendering technique.



(a) Original tile positions. (b) Positions after edge avoidance. (c) Rotated tiles.

Figure 5.12: Orientation and positioning of tiles emphasizes the image edges.

5.6 Discussion and future work

The presented rendering technique uses a novel approach to draw a generic 3-D scene in a mosaic style at interactive framerates. This was not achieved in any previous works. Table 5.1 summarizes framerates for several scenes at various technique settings on Athlon™ 64 3200+ with NVIDIA® GeForce® 7800 GT. Various aspects of the algorithm are exposed via component variables of the individual components, which allows a user to tweak the behavior of the tile positioning, edge avoidance and tile orientation algorithms at run-time to experiment with the technique and to produce the most visually pleasing mosaic.

Screen resolution	Tile count	Scene size (triangle count)		
		772	43492	190990
500 * 500	900	102.0	85.5	69.3
	10000	50.0	44.5	39.5
750 * 750	900	52.8	52.3	41.2
	10000	35.5	33.4	28.5
900 * 900	900	38.0	35.8	31.6
	10000	27.1	25.7	23.4

Table 5.1: Comparison of mosaic renderer framerates.

Instead of using centroidal Voronoi diagrams to determine tile positions, which are hard to compute efficiently, a system of infinitely stiff springs is used. The real-time performance of the spring system comes at the price of a slightly inferior mosaic appeal when compared to the methods using Voronoi diagrams to position the tiles. The spring system has the following limitations:

- It tends to position the tiles in a regular grid pattern, because the individual tiles can not be moved away from their neutral positions on the regular grid

by a considerable distance; such movement would not satisfy the spring constraints.

By using under-relaxation when solving the spring system, the springs stiffness is reduced, which allows the tiles to be positioned in an irregular pattern. However, these irregularities are then allowed both near edges, where they are required, and inside large areas with no edges in their vicinity, where artists usually place the tiles in a more regular pattern.

- The spring system does not respect the tile orientations at all. Therefore the fact that only tiles near the edges are rotated to follow the edge directions is beneficial to the visual appearance of the mosaic. The spring system as proposed would not be able to place tiles in a regular but not axis-aligned pattern properly.

- Tiles of varying shapes and sizes are not supported — all tiles are required to be uniformly sized squares.

- The method is not suitable for animated mosaics. When panning the camera over the scene, the "shower door" effect described by Meier [24] becomes visible. Most of the tiles remain stationary, with the tiles in the vicinity of edges popping from one edge side to another.

The future work should more thoroughly examine the traits of the spring system and determine whether it is possible to eliminate the above mentioned deficiencies by introducing a new set of constraints to the system. For example, a non-constant relaxation coefficient depending on the influence of the nearest edge could be used to make the springs in the vicinity of edges less stiff. Another set of constraints could be used to enforce the relative position of a tile and its neighbors to follow the direction of the nearest edge. The edge pushout algorithm could be also implemented as another set of constraints.

By using a separate spring system for each object in the scene that would track the global screen-space position and orientation of the object, the "shower door" effect visible when panning over the scene or when the objects move could be partially eliminated. However the spring system seems approach to be unsuitable to achieve the quality of the animosaics [31].

Since the edge direction image is generated from automatically computed edge image which is discrete, it can not be as precise as the Hausner's direction field generated from a precise user-defined set of edges. A more work on the algorithm that generates the edge direction image could lead to tile orientations that follow the edges more precisely.

While the mosaic rendering technique is capable of achieving interactive framerates, there is still a room for speed improvements, especially in the `draw_tiles` component which generates the tile geometry. It reads the tile offset and edge direction images and processes them on the CPU. By utilizing the most advanced features, such as accessing textures from vertex programs, described in NVIDIA

whitepaper by Gerasimov et al. [9], the tile geometry could be constructed by a vertex program. Unfortunately, this feature is available only on the NVIDIA GPUs.

Finally, the proposed reduction algorithm that is capable of generating centroidal Voronoi diagrams on the GPU should be further examined. Even though it might be useless for real-time rendering, it would dramatically speed up any algorithm that currently implements the Lloyd's Voronoi diagram centralization on the CPU.

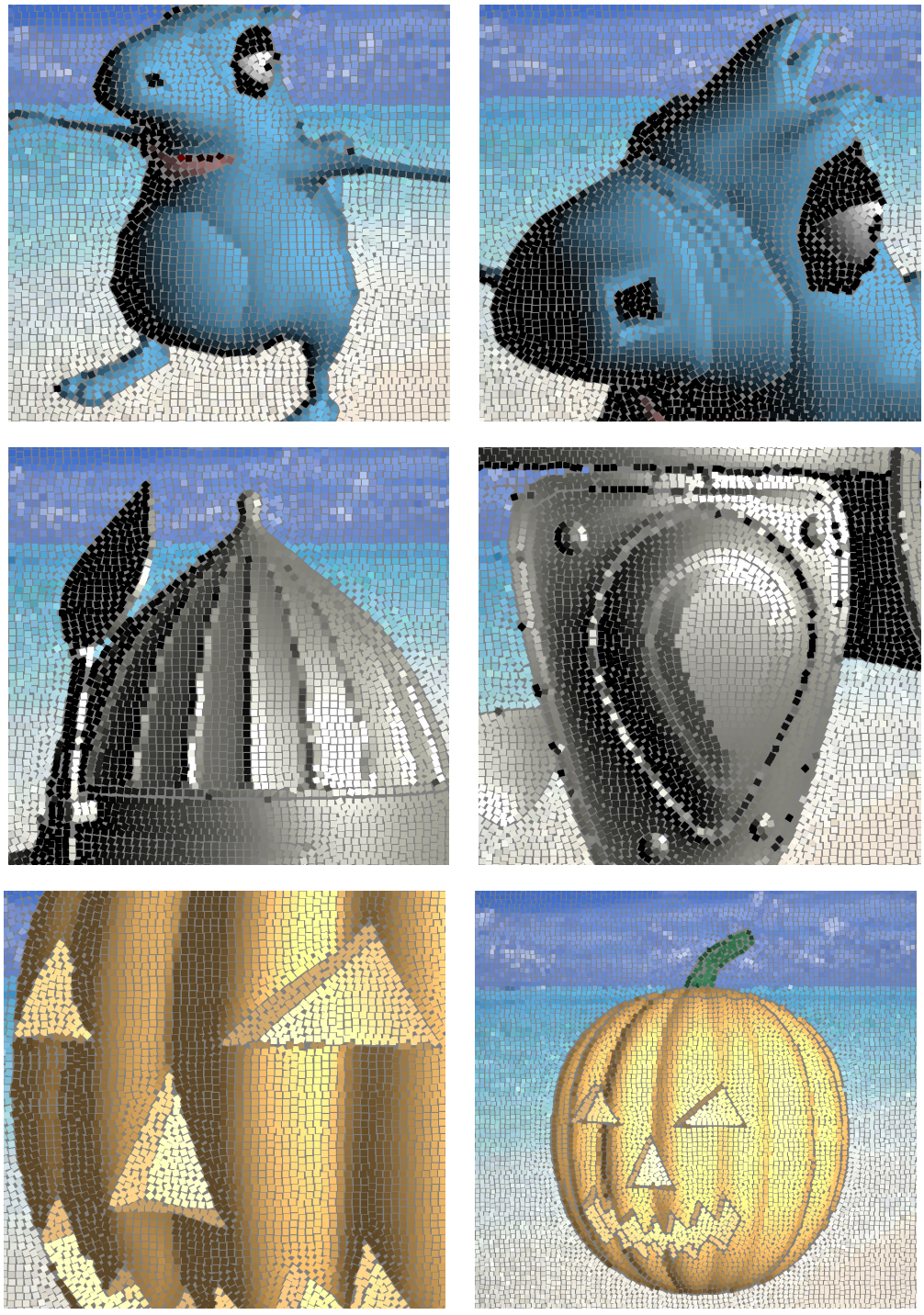


Figure 5.13: Images drawn by the mosaic renderer.

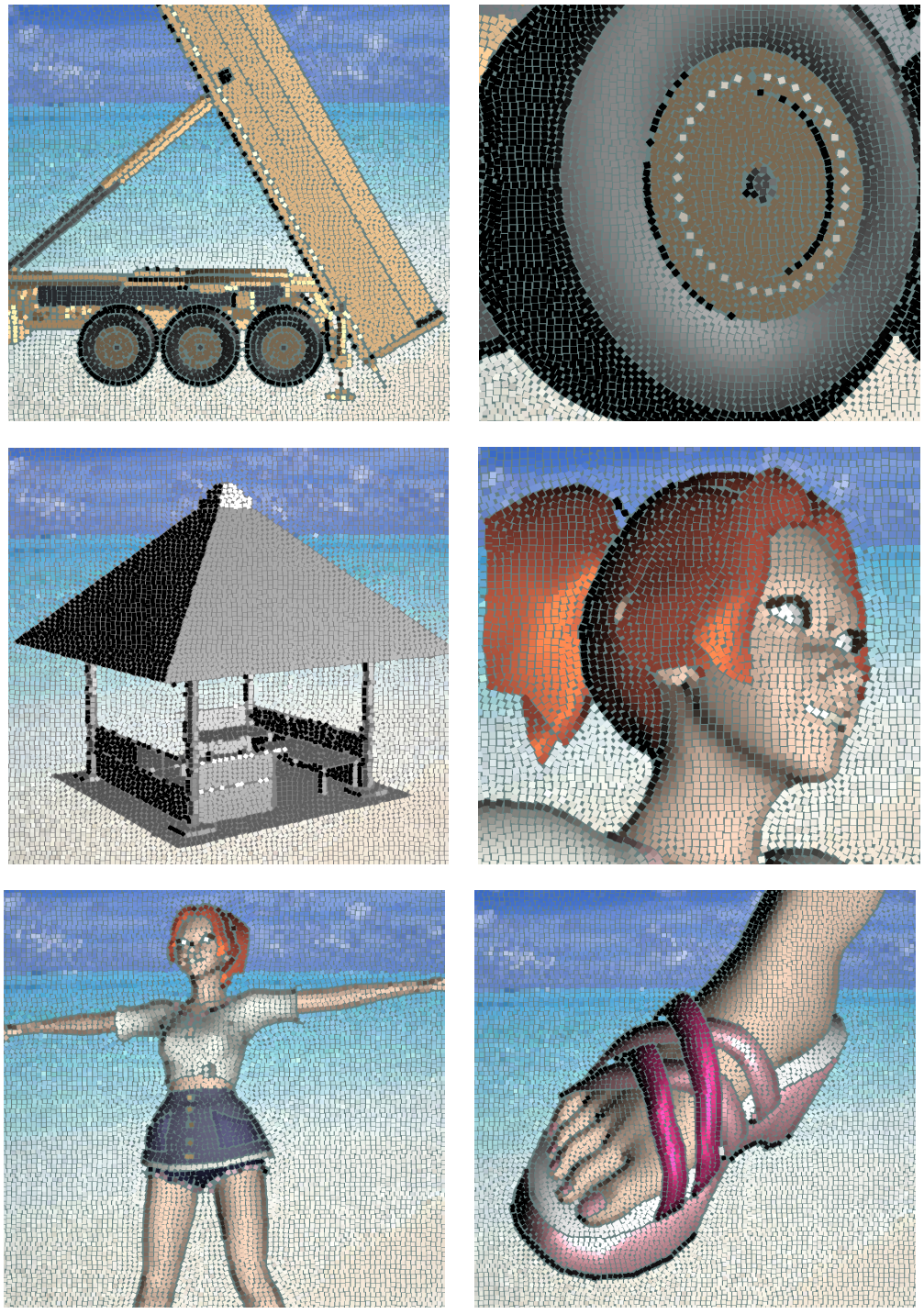


Figure 5.14: More images drawn by the mosaic renderer.

Chapter 6

Conclusions

In this thesis, we presented an application framework suitable for development and testing of real-time rendering techniques implemented on modern programmable GPUs. The framework was used to implement several non-photorealistic rendering methods.

Three image-space edge rendering techniques that can be used to draw silhouettes, contours, border edges, creases and material boundaries of generic three-dimensional polygonal models were implemented and thoroughly examined. Compared to previous implementations, the precision of the depth-based edge detector was improved by using the high-precision depth buffer. By detecting discontinuities in the Laplacian of the depth image instead of discontinuities in the gradient, artifacts at polygons nearly perpendicular to the projection plane were eliminated. We also presented that the combination of all three implemented edge detectors is needed to successfully detect all important edges in the image.

The implemented cartoon shading technique builds on previous works and further improves them. By using more modern GPU programming models with better precision, artifacts at boundaries between step colors were eliminated. Additionally, the implemented cartoon shader supports specular highlights.

We also presented a novel approach to drawing three-dimensional scenes in a mosaic style. The implemented mosaic technique is able to render complex scenes at interactive framerates, which was never achieved before. A GPU-based algorithm to construct centroidal Voronoi diagrams was proposed. The algorithm is much faster than the previously presented solutions, but it is not fast enough for real-time mosaic rendering. To achieve real-time performance, a simulation of a system of infinitely stiff springs was used instead of Voronoi diagrams to position the mosaic tiles.

The future work should focus on the mosaic renderer. The implemented technique still runs partially on the CPU and could be optimized. The tile positioning computed by the spring system simulation gives satisfactory results, but is not as visually appealing as the images produced by methods based on Voronoi diagrams. We analyzed the limitations of the positioning based on the spring system

and suggested several changes to further improve the appearance of the generated mosaics.

The mosaic technique is also not suitable for rendering of animated mosaics, as it exhibits the "shower door" effect. This problem could be partially solved by using separate spring systems for different objects, but more work is needed to achieve reasonable results.

Finally, while the proposed algorithm to compute centroidal Voronoi diagrams was not used in the mosaic renderer, it improves the previously presented methods by performing all steps of the Lloyd's centralization algorithm on the GPU, without the need to copy the generated Voronoi diagrams to CPU in each iteration. The approach presented in this thesis can be used to speed up all methods that use the Lloyd's algorithm.

Bibliography

[1] John W. Buchanan and Mario C. Sousa. The edge buffer: a data structure for easy silhouette rendering. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering (NPAR'00)*, pages 39–42, 2000.

[2] Johan Claes, Fabian Di Fiore, Gert Vansichem, and Frank Van Reeth. Fast 3D cartoon rendering with improved quality by exploiting graphics hardware. In *Proceedings of Image and Vision Computing New Zealand (IVCNZ'01)*, pages 13–18, 2001.

[3] Ketan Dalal, Allison W. Klein, Yunjun Liu, and Kaleigh Smith. A spectral approach to NPR packing. In *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering (NPAR'06)*, 2006.

[4] Philippe Decaudin. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, June 1996.

[5] Philippe Decaudin. *Modélisation par Fusion de Formes 3D pour la Synthèse d'Images – Rendu de Scènes 3D imitant le Style 'Dessin Animé.* Thèse de doctorat, Université de Technologie de Compiègne, France, December 1996.

[6] Sim Dietrich. GPU toon shading. Game Developer's Conference 2000.

[7] Sébastien Dominé, Ashu Rege, and Cem Cebenoyan. Real-time hatching (tribulations in). Game Developer's Conference 2002, March 2002.

[8] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Walk-through illustrations: Frame-coherent pen-and-ink style in a game engine. *Computer Graphics Forum*, 20(3):184–91, 2001.

[9] Philipp Gerasimov, Randima Fernando, and Simon Green. Shader model 3.0: Using vertex textures, 2004.

[10] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'98)*, pages 447–452, 1998.

[11] Paul Haeberli. Paint by numbers: Abstract image representations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'90)*, pages 207–214, 1990.

[12] Mark Harris. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 31: Mapping Computational Concepts to GPUs. Addison-Wesley, 2005.

[13] Alejo Hausner. Simulating decorative mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'01)*, pages 573–580, 2001.

[14] Mike D. Heath, Sudeep Sarkar, Thomas Sanocki, and Kevin W. Bowyer. Comparison of edge detectors: A methodology and initial study. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'96)*, pages 143–148, 1996.

[15] Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. A developer's guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications*, 23(4):29–37, 2003.

[16] Vishvananda Ishaya. Real-time cartoon rendering with DirectX 8.0 hardware. GameDev.net, 2003.

[17] Thomas Jakobsen. Advanced character physics, 2001.

[18] Scott F. Johnston. Lumo: illumination for cel animation. In *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering (NPAR'02)*, pages 45–52, 2002.

[19] III Kenneth E. Hoff, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'99)*, pages 277–286, 1999.

[20] Junhwan Kim and Fabio Pellacini. Jigsaw image mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'02)*, pages 657–664, 2002.

[21] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3D animation. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering (NPAR'00)*, pages 13–20, 2000.

[22] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[23] Lee Markosian, Michael A. Kowalski, Daniel Goldstein, Samuel J. Trychin, John F. Hughes, and Lubomir D. Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'97)*, pages 415–420, 1997.

[24] Barbara J. Meier. Painterly rendering for animation. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH '96)*, pages 477–484, 1996.

[25] Jason L. Mitchell, Chris Brennan, and Drew Card. Real-time image-space outlining for non-photorealistic rendering. SIGGRAPH 2002 Sketch, San Antionio, July 2002.

[26] Masahiro Mori. The uncanny valley. *Energy*, 7(4):33–35, 1970.

[27] Marc Nienhaus and Jürgen Döllner. Edge-enhancement – an algorithm for real-time non-photorealistic rendering. *Journal of WSCG*, 11(2):346–353, 2003.

[28] Guodong Rong and Tiow-Seng Tan. Jump flooding in GPU with applications to voronoi diagram and distance transform. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games (I3D'06)*, pages 109–116, 2006.

[29] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'90)*, pages 197–206, 1990.

[30] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004)*. Silicon Graphics, Inc., 2004.

[31] Kaleigh Smith, Yunjun Liu, and Allison Klein. Animosaics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'05)*, pages 201–208, 2005.

[32] Mario Costa Sousa and Przemyslaw Prusinkiewicz. A few good lines: Suggestive drawing of 3D models. *Computer Graphics Forum*, 22(3):381–390, 2003.

[33] Djemel Ziou and Salvatore Tabbone. Edge detection techniques – An overview. *International Journal of Pattern Recognition and Image Analysis*, 8(4):537–559, 1998.