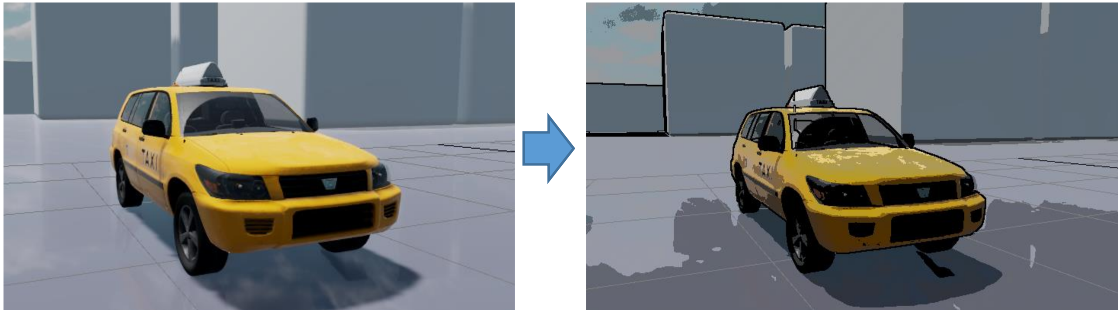
Toon (or “Cel”) Shaders in Stingray

Stingray’s standard rendering pipeline is a powerful system for photorealistic real time rendering. However there are many games where photorealism is not desired. In this tutorial we will examine implementation of a toon shader (or cel shader).

Toon shading is a visual effect that makes a game scene look like a cartoon:



What we see here is actually two separate effects:

1) Outline marking. The taxi and background objects are drawn with black outlines around their edges.
2) Color quantization. Instead of smooth color transitions along the surface of the ground and taxi, there are abrupt jumps between discrete “bands” of color.

Both will be implemented in Stingray’s deferred rendering pipeline as post-process effects. Knobs to control various shader parameters will be exposed in the Stingray Editor UI and passed to the shader programs in a constant buffer to fine tune the effect and explore debug images.

There are multiple techniques for achieving both of these effects. Detailed examination of all of them is beyond the scope of this tutorial however a brief overview of the most popular methods is below.

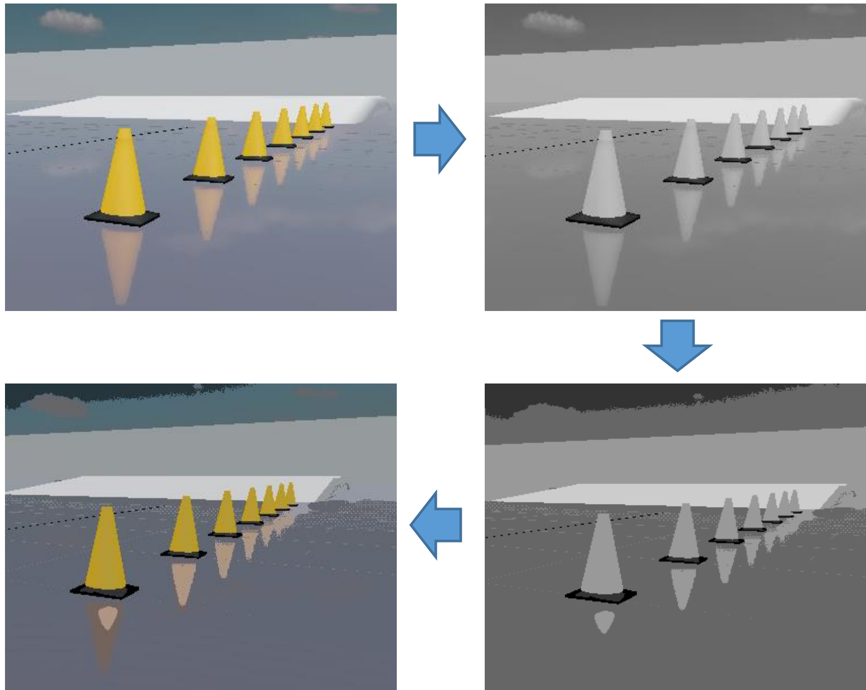
Color Quantization:

In a simple forward rendering pipeline, the pixel shader will calculate the diffuse lighting intensity as a function of the angle between the incoming light vector and the surface normal at the point the pixel shader is processing. The intensity is then used to scale the underlying surface color, typically read from a texture. Rounding the intensity value to, say, the nearest 0.1 will create a banded effect. A general formula is:

    intensity = floor(intensity * num_bands) / num_bands

Because Stingray’s physically based lighting model involves much more complex calculations than the above diffuse lighting calculation, this approach cannot be applied directly. This approach also has the disadvantage that it will not work on the results of post-processing effects applied after the main rendering pass such as bloom.

Instead of reaching into the middle of the rendering pipeline and changing the intensity calculation, we can begin with the final rendered image and iterate over each pixel. The intensity value can be recovered from the color value, quantized, and then applied back into the pixel. By working backwards from the final color value, we avoid the problems with the approach in (a) and we do not have to consider the complexity of the preceding rendering pipeline.



Clockwise from upper left: initial color buffer; intensity values; intensity quantized into 5 discrete bands; final result with toon quantization.

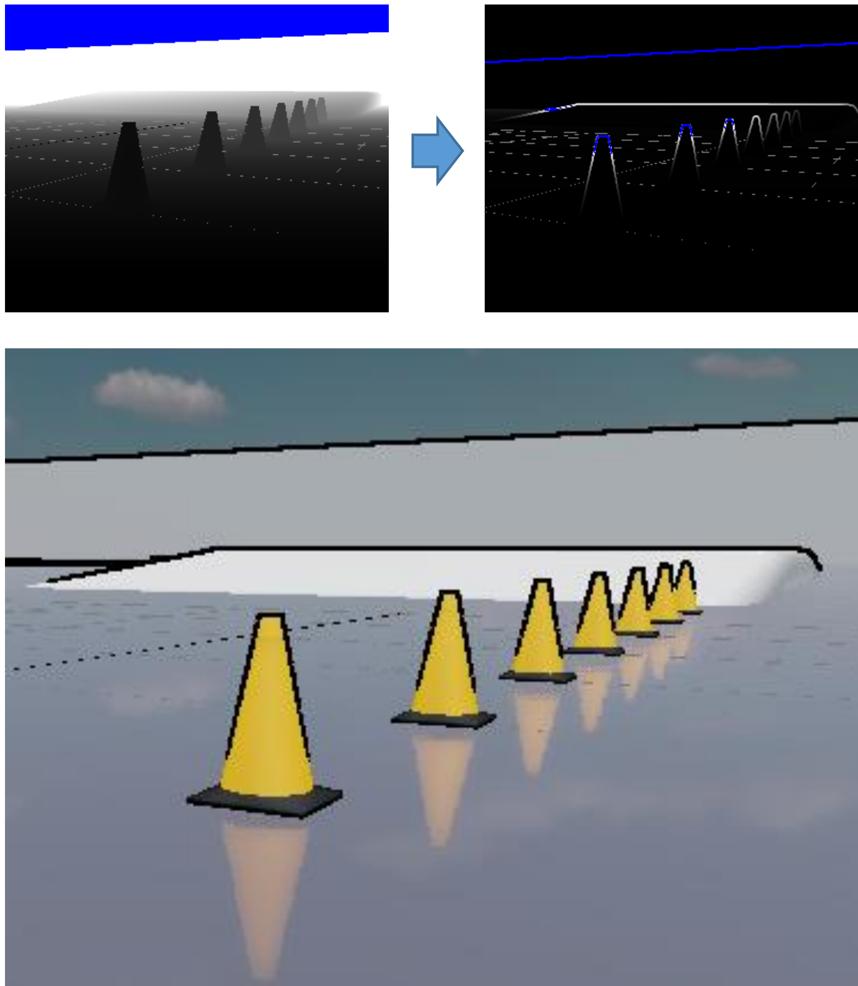
Outline Marking:

A straightforward geometric technique is to render each object twice. First render a slightly enlarged version of the object in all black, then render the object in color on top. A black outline will be left behind. This technique can be implemented with a vertex shader that pushes each vertex slightly in the direction of the normal, which will typically be away from the center of the object. While computationally expensive for complex objects, this approach is well suited to a forward-rendering pipeline and does not require allocation of additional buffers.

Since Stingray uses deferred rendering, image processing edge-detection techniques are a better fit. There are a variety of 2D filters that can be run on an image to detect edges. In particular the Sobel operator performs well for our application. Wikipedia (https://en.wikipedia.org/wiki/Sobel_operator) explains the Sobel operator in detail, but the underlying principle is simple: compare neighboring pixel values and look for sudden changes.

Sobel Filter on Depth Buffer:

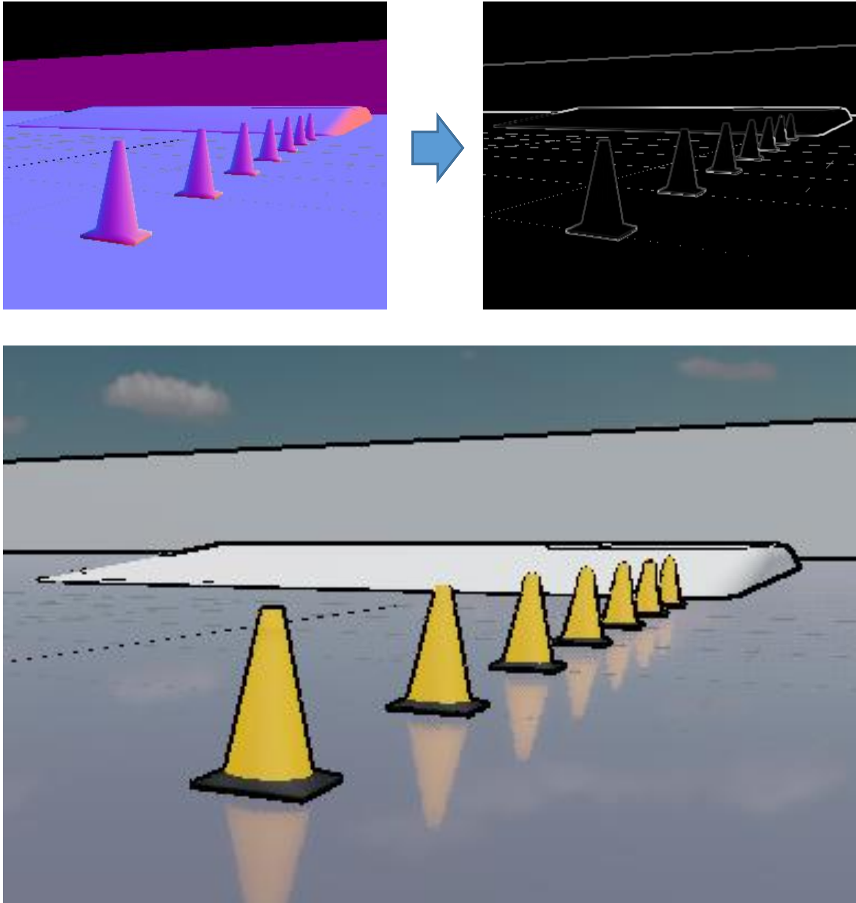
The most obvious place to search for gradients is in the depth buffer: as we “fall off” the edge of an object there will be a sudden increase in the depth value as we hit the object in the background.



Clockwise from upper left: depth buffer, scaled from 3 to 35, depth values greater than 100 in highlighted in blue; output of Sobel filter divided by average depth, scaled from 0 to 15, values greater than 15 highlighted in blue; output image with detected edges (threshold 0.9) marked in black.

Sobel Filter on Normal Buffer:

While edges on the top of the construction cone are distinct, the edges along the bottom were not visible in the depth buffer gradients. This is not surprising since there is little change in depth from the bottom of the cone to the floor. However there is a change in slope which can be detected by searching the normal buffer for gradients:

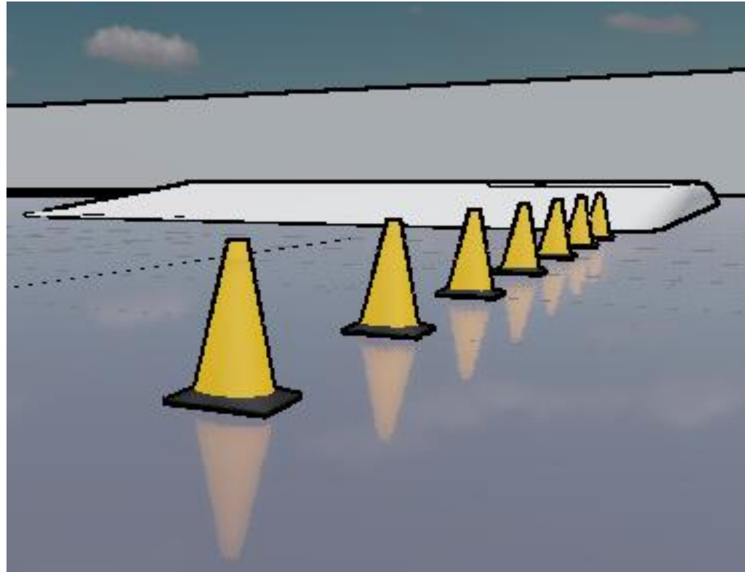


Clockwise from upper left: normal buffer; Sobel filter applied to normal buffer, scaled from 0 to 5; output image with detected edges (threshold 0.9) marked in black.

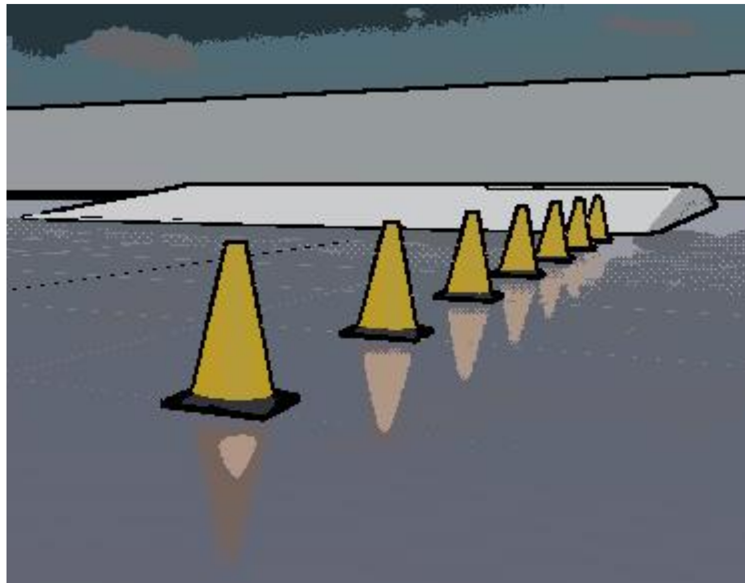
Note that surface normal are vectors and the Sobel operator works on scalar fields. We compute the dot product of each normal vector against it's neighbors as input to the Sobel filter. As expected, this detects the edges on the bottom of the cones. It also misses the edges on the tops of some of the cones further back against the white bump in the background.

The normal buffer is also useful for detecting edges internal to an object such as the line between a character's head and neck.

Both methods of edge detection perform well in certain areas of the scene. By triggering on both normal and depth gradients we obtain a complete outline:



Finally, combining with the color quantization effect we have a complete toon shader:



Implementation in Stingray:

The example toon shader can be downloaded from:

http://forums.autodesk.com/t5/stingray/stingray-how-to-make-a-toon-cel-shader-video/td-p/6012196

Since this demo only affects the core renderer, we can modify those files directly and do not need to import a specific project. Compare the files in the download and patch in changes made to:

```
core/stingray_renderer/renderer.render_config
core/stingray_renderer/shader_libraries/post_processing.shader_source
```

Add the new files:

```
core/stingray_renderer/shading_environment_components/toon.component
core/stingray_renderer/shading_environment_components/toon.shading_environment_mapping
```

Now open the Stingray Editor and create a new project based on a template of your choice. The images above were made using the Vehicle demo.

The ZIP file contains an “orig” directory and “patch” directory which can be diffed to highlight the specific changes made.

Changed Files:

render.render_config: Core rendering configuration file. Allocates buffers and defines format (RGBA8, stencil, floating point, etc.). Defines per-frame render flow as series of shader calls to populate those buffers.

toon.component: defines a Stingray “component” which encapsulates user-defined variables such as the edge color and threshold. This component can be added into the rendering environment to activate the shader.

toon.shading_environment_mapping: mapping between variables defined in toon.component and names that will be made visible to the renderer and shader via the constant buffer.

post_processing.shader_source: raw source code for shader, texture sampler definitions and shader compilation directives.

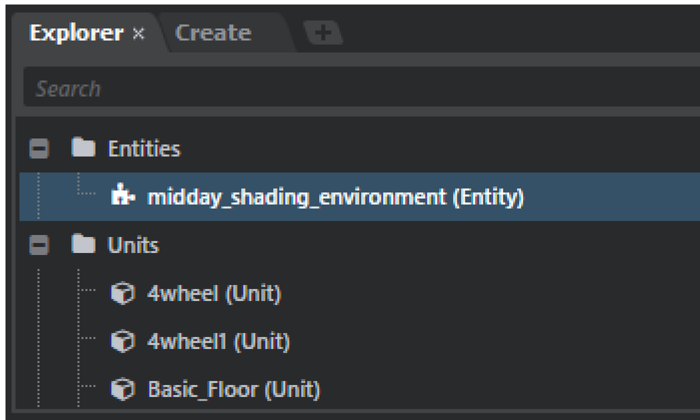
For more information on render configuration files, please see our related video, Stingray Render Config Tutorial: https://forums.autodesk.com/t5/stingray/stingray-render-config-tutorial/td-p/5862052

Shader Environment Components and Variables:

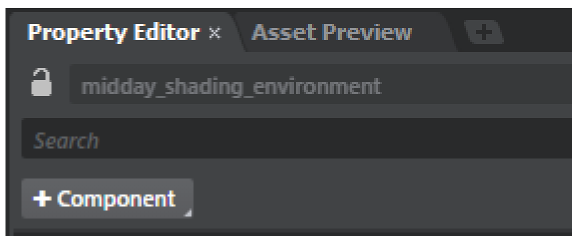
Open any level in the project and view the scene. Among the objects in the level, there is a shading environment which contains various parameters related to effects such as fog and lighting. The “toon.component” defines a new component for our shading environment. Let’s add it to the level to enable the toon shader:

Step #1: Open any level in the project

Step #2: Click on the “midday_shading_environment” item in the Explorer panel

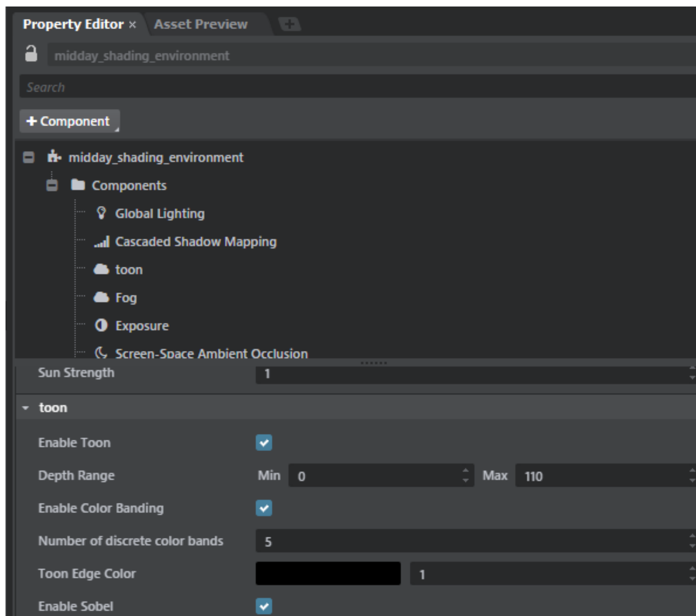


Step #3: Choose “+ Component” in the Property Editor panel



Step #4: Select the “toon” component. (If not present, confirm that the toon.component and toon.shading_environment_mapping files were correctly integrated into core/stingray_renderer/shading_environment_components.)

The toon shader should immediately take effect. Click on the “midday_shading_environment” object and scroll down in the properties panel until the Toon section is found:



“Enable toon shading” checkbox is checked by default. The toon.component file defines this knob:

```
toon_enabled = {
    type = ":bool"
    default = true
    editor = {
        label = "Enable Toon"
        description = "Is Toon Enabled?"
    }
}
```

By default toon_enabled is true which is why the shader took effect as soon as we loaded the Toon component.

Grepping the core/stingray_renderer tree for “toon_enabled” reveals that it is used in renderer.render_config:

```
{ type="dynamic_branch" shading_environment={ toon_enabled=true }
       pass = [
              { type="fullscreen_pass" shader="copy" input=["output_target"]
       output=["postfxdemo_scratch"] }
              { type="fullscreen_pass" shader="toon" input=["postfxdemo_scratch"
       "depth_stencil_buffer"] output=["output_target"] }
       ]
}
```

This “dynamic branching” feature causes the renderer to invoke the toon shader when the enable knob in the GUI is checked. The Toon panel in the shader properties exposes many other knobs useful for debugging and tuning the shader and are demonstrated in the companion video on our YouTube channel: https://youtu.be/I54fy7MB_8g

Shader Programs:

Shader environment variables can also be referenced directly in shader programs via the constant buffer. Search for “toon” in post_processing.shader_source and locate the CBUFFER:

```
// Constant buffer
// Stingray will populate these constants before invoking our shader.
// The engine identifies constants based on their names. For example,
// any float2 with the format inv_input_textureN_size will be set to
// (1/texture_width, 1/texture_height).
CBUFFER_START(c0)

float4x4 world_view_proj;

// (1/texture_width, 1/texture_height). Used to calculate offsets to adjacent pixels.
float2 inv_input_texture0_size;

// Rest of the buffer are debug knobs provided by the GUI. To understand how these
// knobs are exposed grep core/stingray_renderer/* for "toon_edge_color".
//
// The file core/stingray_renderer/shading_environment_components/toon.component
// makes the variables visible in the GUI, assigning labels like "Is Toon Enabled?"
// core/stingray_renderer/shading_environment_components/toon.shading_environment_mapping
// maps the GUI variables to constant names and is what the Stingray renderer uses to match
// the inputs to the named constants below.
```

```
// Only apply the toon effect to pixels within this linearized depth range
UNIFORM float2 toon_depth_range;

// The edge color. Black is a common choice, but any color can be used.
UNIFORM float3 toon_edge_color;

// We encode multiple scalar values into each component of the two below vectors:
//   toon_float_param.x:  exponent to apply to result of edge detection filter, typically 1 to 1.5
//   toon_float_param.y:  cutoff threshold for edge detection filter
//   toon_float_param.z:  number of color bands to quantize colors into, 3 - 10 are typical values
//   toon_float_param.w:  distance (in pixels) to offer the samples taken, 1.0 is a typical value
//   toon_normal_param.x: cutoff threshold for the edge detection filter when applied to normal buffer
//   toon_normal_param.y: distance (in pixels) to offset the samples taken for the edge detection filter
//                        on the normal buffer. Typically set to 1 to sample the neighboring pixels. Can
//                        experiment with fractional values as linear interpolation is used.
UNIFORM float4 toon_float_param;
UNIFORM float2 toon_normal_param;

// Enables us to toggle different components of the toon shader on and off to better
// see their effect.
UNIFORM bool toon_enable_color_banding;
UNIFORM bool toon_enable_sobel_edge_detection;
UNIFORM bool toon_enable_normal_edge_detection;

// We can view certain intermediate values graphically to assist with debugging and choosing
// appropriate threshold values. See the code for the meaning of each of these.
UNIFORM bool toon_debug_output_depth;
UNIFORM bool toon_debug_output_sobel;
UNIFORM bool toon_debug_output_sobel_scaled;
UNIFORM bool toon_debug_output_normal_gradient;
UNIFORM bool toon_debug_output_luminance;
UNIFORM bool toon_debug_output_discretized_luminance;

// Unlike color values which vary from 0.0 to 1.0 in each component, the above debug outputs
// can have a much wider range. toon_debug_range specifies a range to scale the values to. For
// example, a range of [0, 50] is typically good for viewing linearized depth values.
// Any pixels outside of toon_debug_band will be marked green or blue and is useful for establishing
// distance of certain objects when viewing a depth buffer.
UNIFORM float2 toon_debug_range;
UNIFORM float2 toon_debug_band;

CBUFFER_END
```

The toon.shader_environment_mapping file provides a mapping from the knobs exposed in the GUI and the shader constant buffer. To add your own constants:

1) Create an entry in toon.component with the definition to be exposed in the GUI.
2) Create an entry in toon.shader_environment_mapping to map the variable to a named shader constant.
3) Declare the constant with a matching name in the CBUFFER section of your shader program. Before invoking your shader, Stingray will load the constant buffer with the appropriate value.

Let’s now review the shader program itself. It is invoked during through the post processing resource generator in render.render_config:

```
{ type="fullscreen_pass" shader="copy" input=["output_target"]
output=["postfxdemo_scratch"] }
{ type="fullscreen_pass" shader="toon" input=["postfxdemo_scratch"
"depth_stencil_buffer"] output=["output_target"] }
```

(For more details on resource generators, please see our previous tutorial at https://www.youtube.com/watch?v=xq22zjb8eB8)

The toon shader is passed two input buffers as arguments: a copy of the final output buffer and the depth buffer. These will be accessed via the texture samplers defined in the shader program which lives in post_processing.shader_source:

```
hlsl_shaders = {
	toon = {
		includes = [ "common", "gbuffer_access" ]
		samplers = {
			input_texture0 = { sampler_states = "clamp_point" }
			input_texture1 = { sampler_states = "clamp_point" }
			gbuffer0 = { sampler_states = "clamp_linear" }
			gbuffer1 = { sampler_states = "clamp_linear" }
			gbuffer2 = { sampler_states = "clamp_linear" }
			gbuffer3 = { sampler_states = "clamp_linear" }
		}
```

…

```
DECLARE_SAMPLER_2D(input_texture0); // Color buffer. We will modify to apply toon effect.
DECLARE_SAMPLER_2D(input_texture1); // Depth buffer. Used to locate edges.
DECLARE_SAMPLER_2D(gbuffer0);       // We only use the normals in the guffer, but bring
DECLARE_SAMPLER_2D(gbuffer1);       // all gbuffer components to simplify
DECLARE_SAMPLER_2D(gbuffer2);       // experimentation.
DECLARE_SAMPLER_2D(gbuffer3);
```

We can also bring in the gbuffer without explicitly passing it as an argument because our shader includes “gbuffer_access”.

Since we’re accessing individual pixels in the color and depth buffers, no interpolation is desired and we choose point sampling. Linear interpolation is used for accessing normal values from the gbuffer.

Notice that the shader block contains three code sections: “vp_code,” “fp_code” and “code.” For this demo, we only implemented “code” and left “vp_code” and “fp_code” unmodified from a blur shader copied as a template. Stingray supports both DirectX and OpenGL based platforms, and requires shader code be defined for both. As we are demoing on Windows only the DirectX portion was written.

Moving on to the actual shader code we see how color quantization is performed:

```
static const half3 luminance_vector = half3(0.2127, 0.7152, 0.0721);

// Apply color quantization
if(toon_enable_color_banding >= 1.0)
{
	// Recover luminance from the color buffer value
	float luminance = dot(c.rgb, luminance_vector);

	if(toon_debug_output_luminance >= 1.0)
		return float4(luminance, luminance, luminance, 1.0);

	// Discretize the luminance value into toon_num_color_bands
	float discretized_luminance = floor(luminance * toon_num_color_bands) / toon_num_color_bands;

	if(toon_debug_output_discretized_luminance >= 1.0)
		return float4(discretized_luminance, discretized_luminance, discretized_luminance, 1.0);

	// Adjust the color to match the discretized luminance value
```

```
        c.rgb /= luminance;
        c.rgb *= discretized_luminance;
}
```

Notice that if the debug_output_luminance and output_discretized_luminance flags are checked in the GUI, the shader will display debug output. In the code snippets below debugging has been removed for brevity.

The Sobel edge detector is implemented by convolving two kernels with the depth buffer, one to detect gradients in the X direction and another in the Y direction. The results are combined to determine the relative amount of local change at each pixel. Pixels that are above a threshold are marked as edges.

```
// Sobel filter kernel. See https://en.wikipedia.org/wiki/Sobel_operator
const static float3 sobel_x[6] = {
        float3(-1, -1, -1), // third element is weight
        float3(-1,  0, -2),
        float3(-1,  1, -1),
        float3( 1, -1,  1),
        float3( 1,  0,  2),
        float3( 1,  1,  1)
};

const static float3 sobel_y[6] = {
        float3(-1, -1, -1), // third element is weight
        float3( 0, -1, -2),
        float3( 1, -1, -1),
        float3(-1,  1,  1),
        float3( 0,  1,  2),
        float3( 1,  1,  1)
};

// Calculate the result of the Sobel operator on this pixel. Also track the minimum (closest)
// linearized depth value encountered. Typical test scenes had a value of 5 - 20 for nearby
// objects.
float2 offset_multiplier = inv_input_texture0_size * float2(toon_depth_offset, toon_depth_offset);
float weighted_sum_sobel_x = 0;
float lowest_depth = 1000000.0;
for (i = 0; i < 6; ++i) {
        float depth = TEX2D(input_texture1, input.uv + float2(sobel_x[i].x, sobel_x[i].y) *
offset_multiplier).r;
        depth = linearize_depth(depth);
        lowest_depth = min(depth, lowest_depth);
        weighted_sum_sobel_x += sobel_x[i].z * depth;
}
float weighted_sum_sobel_y = 0;
for (i = 0; i < 6; ++i) {
        float depth = TEX2D(input_texture1, input.uv + float2(sobel_y[i].x, sobel_y[i].y) *
offset_multiplier).r;
        depth = linearize_depth(depth);
        lowest_depth = min(depth, lowest_depth);
        weighted_sum_sobel_y += sobel_y[i].z * depth;
}

// Now we have the magnitude of the gradient.
float gradient = sqrt((weighted_sum_sobel_x * weighted_sum_sobel_x) + (weighted_sum_sobel_y *
weighted_sum_sobel_y));

// The depth difference between two adjacent pixels on a flat plane
// far in the distance will be much greater than for the part of the plane close to the viewer.
// Let's scale by the minimum depth value to account for this to avoid false triggers on far away objects.
float scaled_gradient = gradient / lowest_depth;

// Also scale it with an exponent. This changes the distribution within [0, 1] so that
// values over a threshold of, say, 0.7 will decrease slightly below the threshold.
// Higher exponent means fewer pixels classified as edge. Values of 1.0 to 1.5 work
```

```
// well for the exponent. This step is not strictly necessary and could be removed as
// an optimization.
scaled_gradient = pow(scaled_gradient, toon_exponent);

// Parts of the scene that are far away will generate too many false positives due to a rapid
// change in depth. For example a flat plane far in the distance will suffer from this effect.
// Don't mark these areas with outlines.
if(lowest_depth > toon_depth_range.x && lowest_depth < toon_depth_range.y)
{
	// We're in an area where we're applying toon edges. If the scaled gradient exceeds
	// the threshold, output the edge color.
	if(scaled_gradient > toon_edge_threshold)
	{
		c.rgb = toon_edge_color.rgb;
		return c;
	}
}
```

Similar code is applied to the normal buffer and can be found in the source file. The best way to explore the shader programs is to use the debug features in the GUI to look at the output graphically step-by-step as demonstrated in the companion video: https://youtu.be/I54fy7MB_8g All of the intermediate images in this writeup were generated with the debug knobs.

Two practical notes on the shader program parameters:

Sampling offsets: both depth and normal based edge detection exposes a sampling offset parameter. A value of 1.0 will check the immediate neighboring pixels when evaluating the Sobel filter. Larger values will result in thicker edges, sometimes with the unwanted side effect of double lines (or "halos"). Since point sampling is used on the depth buffer, only integer sampling offsets make sense. Linear sampling is used on the normal buffer so smaller values like 0.5 are meaningful.

Exponent: the exponent parameter for Sobel filter on the depth buffer can be used to sharpen the edges just as with the specular exponent common on material properties. It has little effect on the scenes we tested and could be removed as an optimization.

Default values used in screenshots:

Depth Range:	[0, 110]
Color Bands:	5
Edge Color:	Black
Depth Buffer Edge:	Sampling Offset = 1.0, Exponent = 1.5, Threshold = 0.9
Normal Buffer Edge:	Sampling Offset = 1.0, Threshold = 0.8

For any suggestions or questions on this shader, please visit the Autodesk Stingray forum:

http://forums.autodesk.com/t5/stingray/bd-p/800