Chapter 1. Quick Tutorial

Overview

This chapter presents a short PerfHUD 5 tutorial to quickly introduce you to several convenient and powerful new features. Even if you've used previous versions of PerfHUD, we highly recommend that you read through this tutorial because so much is new in PerfHUD 5.

Launching PerfHUD

By default, the PerfHUD installer will place a shortcut to the PerfHUD Launcher on your desktop. To analyze an application, simply drag its icon onto the PerfHUD launcher. Keep in mind that the application needs to opt-in for PerfHUD analysis, to prevent unauthorized parties from analyzing your application.

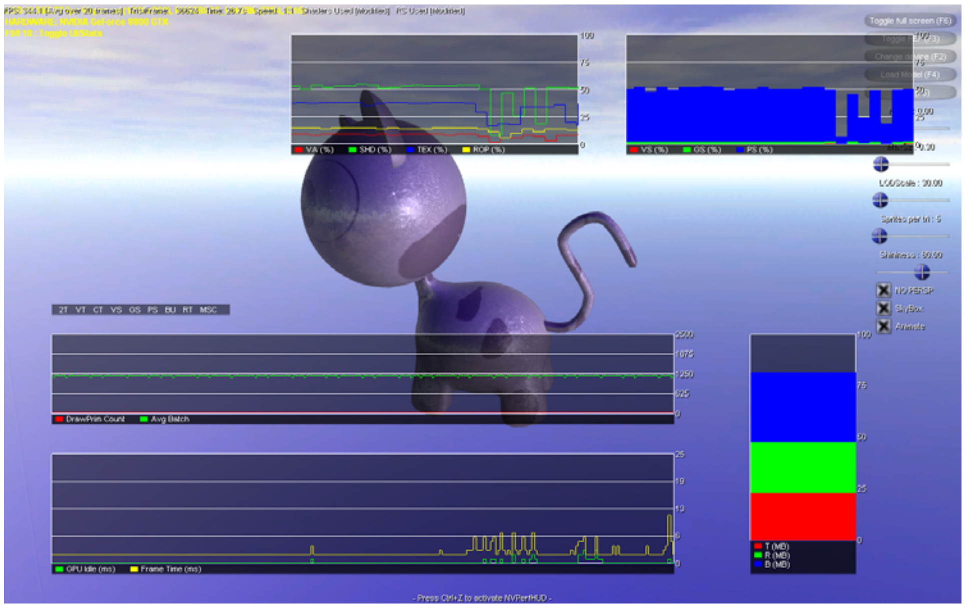
Let's analyze the sample DirectX 10 application that ships with PerfHUD, Sparkles. (This sample is taken from the NVIDIA Direct3D 10 SDK, and includes the opt-in modification.) For this particular application, you can use the "Sparkles Sample" shortcut in the PerfHUD group in the Start menu.

If this is the first time you're running PerfHUD, you'll see a configuration dialog box. The main thing you have to do here is to choose a shortcut key. Pick **Ctrl+Z**.

Once you click **OK**, Sparkles will start, and PerfHUD will be running on it, as shown below.

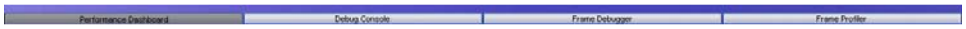
Note that any keyboard or mouse input will still go the Sparkles application, and not to PerfHUD, until you activate PerfHUD using your hotkey (Ctrl+Z). PerfHUD reminds you of your hotkey with a message at the bottom of the screen: "Press Ctrl+Z to activate PerfHUD".

Before activating PerfHUD, press **F9** and **F10** to hide the user interface of Sparkles, reducing clutter. (Remember, these are hotkeys of Sparkles – once PerfHUD is active, F9 and F10 will perform different functions.)



Activating PerfHUD

Activate PerfHUD by pressing **Ctrl+Z**. You'll see the status line at the bottom of the screen change to four buttons, one for each mode of PerfHUD:



Now, any keyboard or mouse input you make will affect PerfHUD. You can toggle between PerfHUD and your application at any time. For example, you may want to navigate to a different part of the scene to analyze it, and then re-enable PerfHUD when you're done.

Help Screen

At any time while you're running PerfHUD, you can press **F1** to view the Help window. This window also has options for getting System Information as well as setting various PerfHUD options.

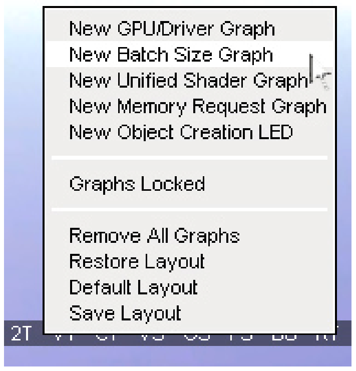
Performance Dashboard

PerfHUD starts in the Performance Dashboard. This mode displays many useful data values, such as per-unit GPU utilization, driver time, memory usage, and more. New in PerfHUD 5 is the ability to completely customize the Performance Dashboard's layout.

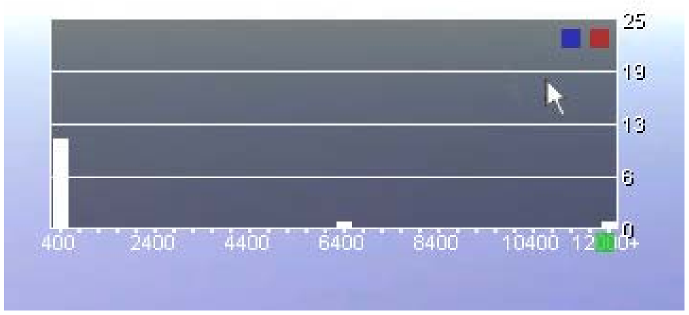
Creating a New Batch Size Graph

Let’s start by creating a new Batch Size graph. This graph displays batches and sizes, allowing you to easily understand the batching characteristics of your application.

To add a new graph, **right-click on the background** and choose **New Batch Size Graph**.



A new Batch Size graph will now appear with its default settings:

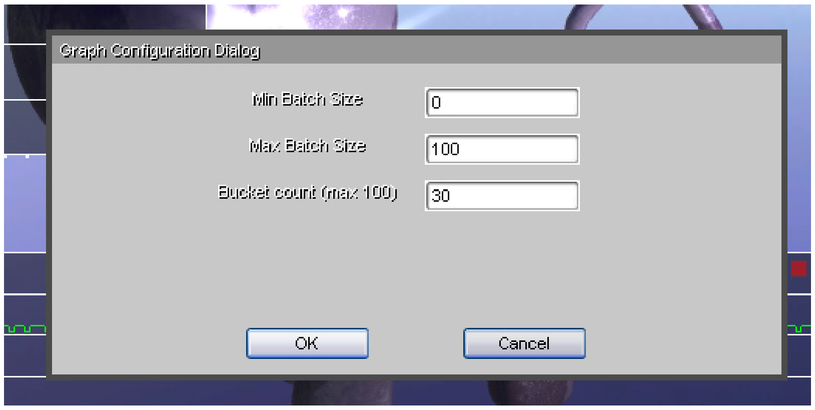


Every graph in the Performance Dashboard is customizable. To do this, simply hover your mouse anywhere on the graph. You’ll see three boxes appear: blue and red boxes at the upper right of the graph, and a green box on the lower right:

- Clicking on the **blue box** brings up a configuration dialog.
- Clicking on the **red box** closes the current graph.
- Clicking and dragging on the **green box** resizes the current graph.

Let’s customize the Batch Size Graph. First, resize it using the **green box**. Then click on the **blue box** and you’ll see the Graph Configuration Dialog.

Set the **Maximum Batch Size** to **100**. Then click **OK**. The graph will now show more bars.

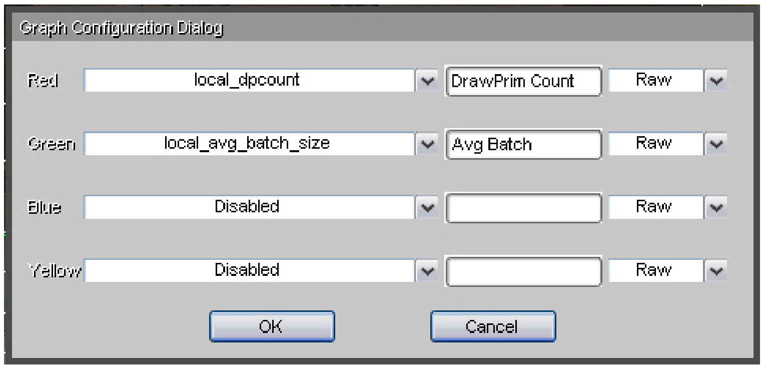


Min Batch Size: 0
Max Batch Size: 100
Bucket count (max 100): 30

Adding Signals

The most common type of graph in PerfHUD is the GPU/Driver Graph. Each GPU/Driver graph can display up to 4 signals simultaneously. PerfHUD 5 allows you to choose from a huge list of both GPU and driver signals, allowing you to monitor virtually any aspect of your application's graphics performance.

Let's add some signals to the GPU/Driver graph that displays the DrawPrimitive Count and Average Batch by default. To do this, hover over the graph and **click on the blue square** at the upper-right of the graph. A Graph Configuration Dialog will pop up:



Red	local_dpcount	DrawPrim Count	Raw
Green	local_avg_batch_size	Avg Batch	Raw
Blue	Disabled		Raw
Yellow	Disabled		Raw

Here, you can choose any signal you want for each line color, as well as descriptions for each. You can also decide whether you want to graph the raw signal or a percentage.

Choose **D3D FPS** for the blue line, and name it "FPS".

Choose **D3D vidmem MB** for the yellow line, and name it "D3D Vid Mem (MB)"

Speeding Up and Slowing Down Time

By pressing the **+** and **–** keys, you can scale the passing of time from 6x faster than normal down to 1/8 speed. Pressing the **–** key again when at 1/8 speed will freeze time completely. Controlling time is helpful when you want to find a particularly troublesome set of frames.

Running Experiments

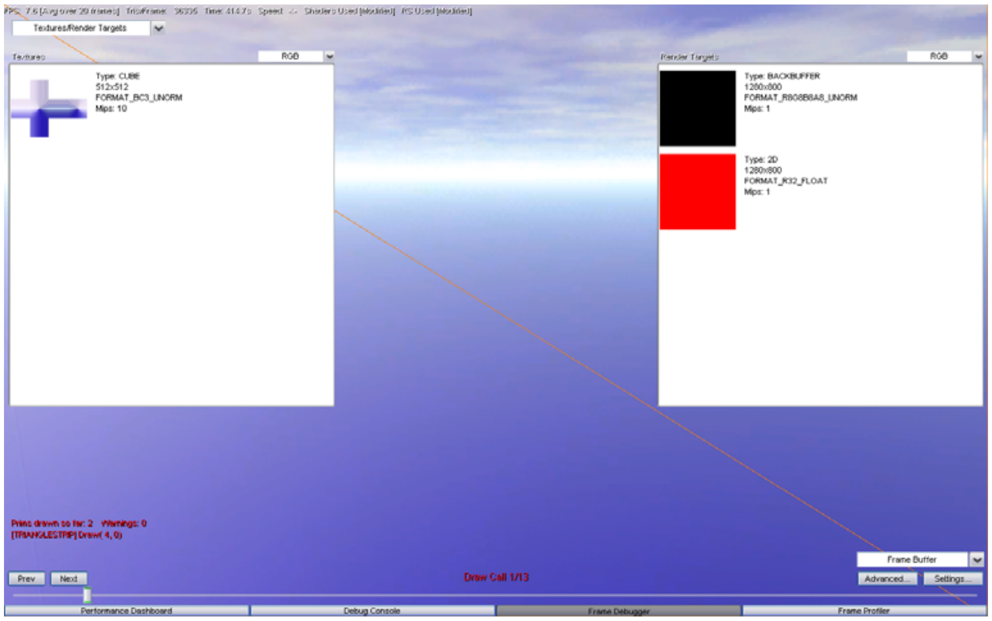
You can also perform various useful experiments from the Performance Dashboard. These are listed below along with their respective keyboard shortcuts.

Use 2x2 Textures	Ctrl+T
Set NULL Viewport	Ctrl+V
Wireframe	Ctrl+W
Ignore Draw Calls	Ctrl+N
Color Fixed Function Shaders Red	Ctrl+1
Color ps_1_1 Shaders Light Green	Ctrl+2
Color ps_1_3 Shaders Green	Ctrl+3
Color ps_1_4 Shaders Yellow	Ctrl+4
Color ps_2_0 Shaders Light blue	Ctrl+5
Color ps_2_a Shaders Blue	Ctrl+6
Color ps_3_0 Shaders Orange	Ctrl+7
Color ps_4_0 Shaders Red	Ctrl+8

Using the Frame Debugger

The Performance Dashboard is most useful for finding a troublesome spot in your scene. Once you've found that spot, you will often want to freeze the frame, debug its draw calls, and analyze its performance in detail.

Press F7 to switch to the Frame Debugger. The Frame Debugger will show you just the first draw call in the scene, which in this case is the skybox:



FPS: 7.6 [Avg over 20 frames] Tris/Frame: 36336 Time: 414.7s Speed: -/- Shaders Used [Modified] RS Used [Modified]

Type: CUBE
512x512
FORMAT_BC3_UNORM
Mips: 10

Type: BACKBUFFER
1280x800
FORMAT_R8G8B8A8_UNORM
Mips: 1

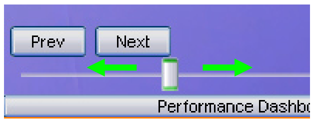
Type: 2D
1280x800
FORMAT_R32_FLOAT
Mips: 1

Prims drawn so far: 2 Warnings: 0
[TRIANGLESTRIP] Draw(4, 0)

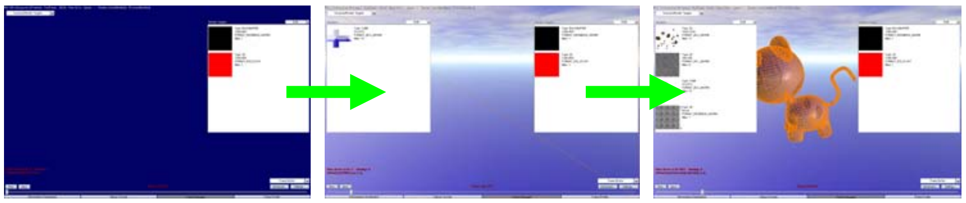
Draw Call 1/13

Scrubbing Through the Frame

Click and drag the slider at the bottom of the screen from side to side.



You'll see how the frame builds up with various draw calls.



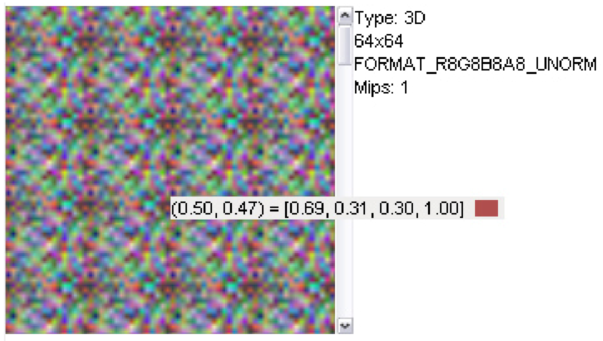
The current draw call is highlighted with an orange wireframe.

You can use the **up** and **down** arrow keys to decrement or increment the current draw call. **Home** jumps to the first draw call, and **End** jumps to the last draw call. **Page Up** and **Page Down** decrement or increment the current draw call by larger amounts.

Drag the slider to draw call 2. You should see the cat highlighted in orange wireframe.

Viewing Textures and Render Targets

All the textures used by the current draw call are shown in the Textures panel on the left of the screen. **Click on the Textures panel** (to get focus) and **press + twice** to enlarge the textures. (Pressing - will reduce the textures.) Note that if you hover over a texture, a tooltip will appear showing u-v coordinates and RGBA color information.



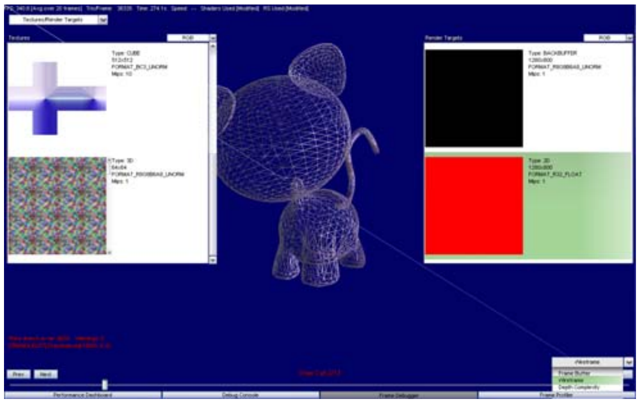
Type: 3D
64x64
FORMAT_R8G8B8A8_UNORM
Mips: 1

(0.50, 0.47) = [0.69, 0.31, 0.30, 1.00]

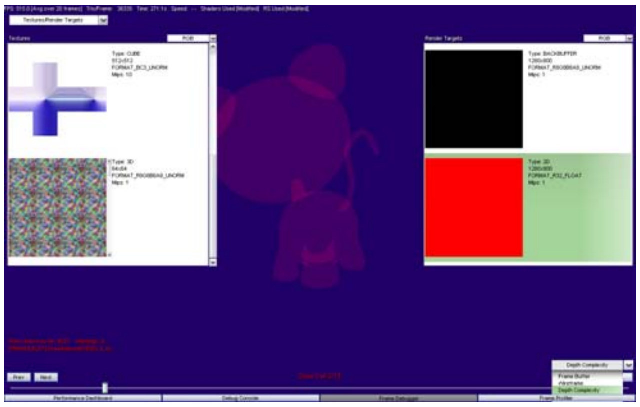
On the right is the list of Render Targets. You can perform the same operations in that panel as in the Textures panel.

Changing the Viewing Mode

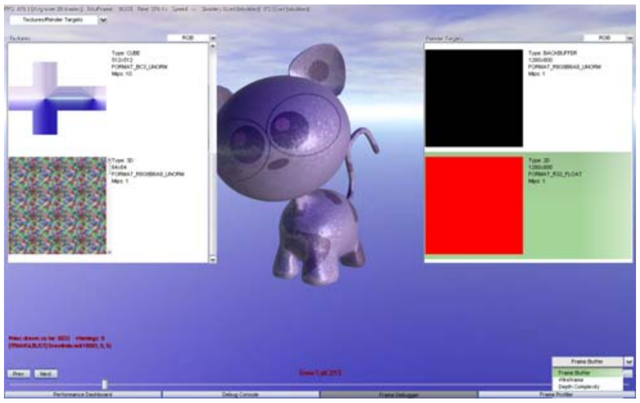
In addition to viewing the **Frame Buffer** as usual, you can also view **Wireframe**, **Depth Complexity**, and **Depth Buffer** renderings for the current frame by choosing options from the drop-down. These views are shown below.



Wireframe



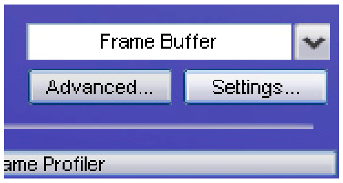
Depth Complexity



Frame Buffer

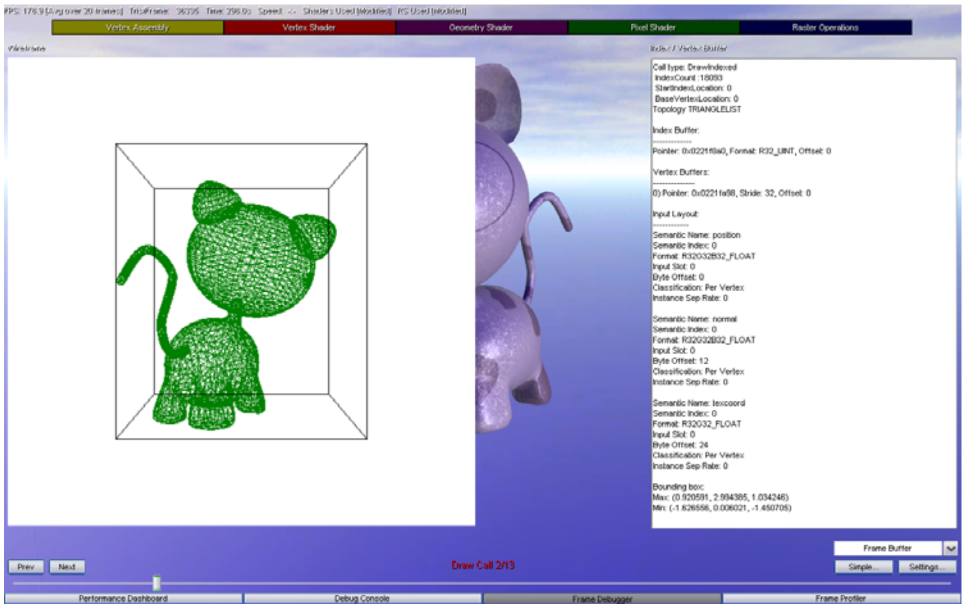
Using the Advanced State Inspectors

To analyze a particular draw call in depth, you can use PerfHUD's Advanced State Inspectors. Access these by clicking on the **Advanced…** button at the lower-right of the screen.



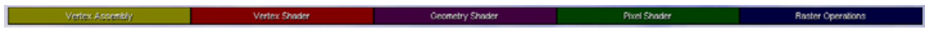
The Vertex Assembly State Inspector

You'll first see the Vertex Assembly State Inspector. Here you can see the geometry used in the current draw call. You can click and drag the mouse on the geometry to rotate it. You can also view details about the geometry in the panel on the right.



```
Call type: DrawIndexed
 IndexCount :18093
 StartIndexLocation: 0
 BaseVertexLocation: 0
Topology TRIANGLELIST

Index Buffer:
--------------
Pointer: 0x0221f8a0, Format: R32_UINT, Offset: 0

Vertex Buffers:
---------------
0) Pointer: 0x0221fa98, Stride: 32, Offset: 0

Input Layout:
-------------
Semantic Name: position
Semantic Index: 0
Format: R32G32B32_FLOAT
Input Slot: 0
Byte Offset: 0
Classification: Per Vertex
Instance Sep Rate: 0

Semantic Name: normal
Semantic Index: 0
Format: R32G32B32_FLOAT
Input Slot: 0
Byte Offset: 12
Classification: Per Vertex
Instance Sep Rate: 0

Semantic Name: texcoord
Semantic Index: 0
Format: R32G32_FLOAT
Input Slot: 0
Byte Offset: 24
Classification: Per Vertex
Instance Sep Rate: 0

Bounding box:
Max: (0.920591, 2.994385, 1.034246)
Min: (-1.626556, 0.006021, -1.450705)
```

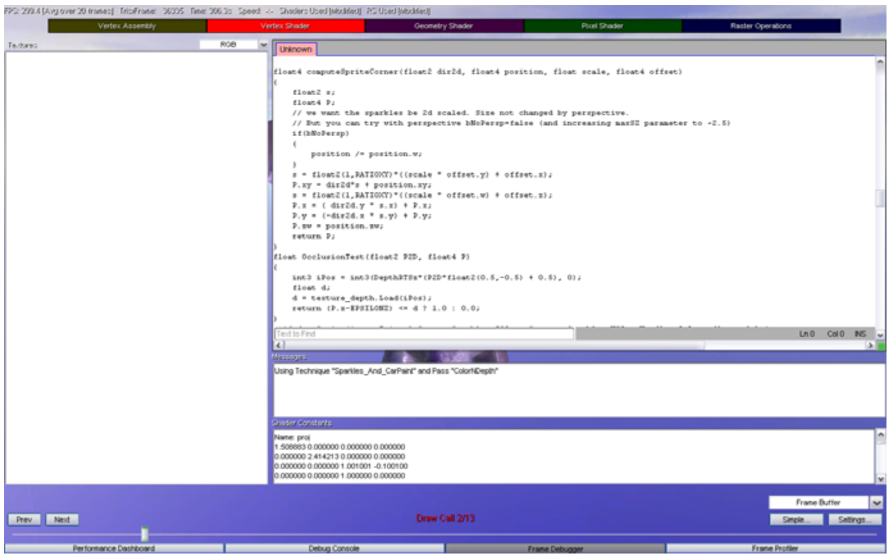
Next, switch to the Vertex Shader state inspector by clicking on the red **Vertex Shader** block at the top of the screen.



Vertex Shader State Inspector

The Vertex Shader State Inspector shows you any vertex shader code from the current draw call, as well as any textures and shader constants that are used. In this case, there are no textures, so the panel at the left of the screen is blank. You can also edit the shader in real-time (we'll cover that when we look at the pixel shader).

Click on the purple **Geometry Shader** block at the top of the screen.



```
float4 computeSpriteCorner(float2 dir2d, float4 position, float scale, float4 offset)
{
    float2 s;
    float4 P;
    // we want the sparkles be 2d scaled. Size not changed by perspective.
    // But you can try with perspective bNoPersp=false (and increasing maxSZ parameter to ~2.5)
    if(bNoPersp)
    {
        position /= position.w;
    }
    s = float2(1,RATIOXY)*((scale * offset.y) + offset.x);
    P.xy = dir2d*s + position.xy;
    s = float2(1,RATIOXY)*((scale * offset.w) + offset.z);
    P.x = ( dir2d.y * s.x) + P.x;
    P.y = (-dir2d.x * s.y) + P.y;
    P.zw = position.zw;
    return P;
}
float OcclusionTest(float2 P2D, float4 P)
{
    int3 iPos = int3(DepthRTSz*(P2D*float2(0.5,-0.5) + 0.5), 0);
    float d;
    d = texture_depth.Load(iPos);
    return (P.z-EPSILONZ) <= d ? 1.0 : 0.0;
}
```

```
Using Technique "Sparkles_And_CarPaint" and Pass "ColorNDepth"
```

```
Name: proj
1.508883 0.000000 0.000000 0.000000
0.000000 2.414213 0.000000 0.000000
0.000000 0.000000 1.001001 -0.100100
0.000000 0.000000 1.000000 0.000000
```

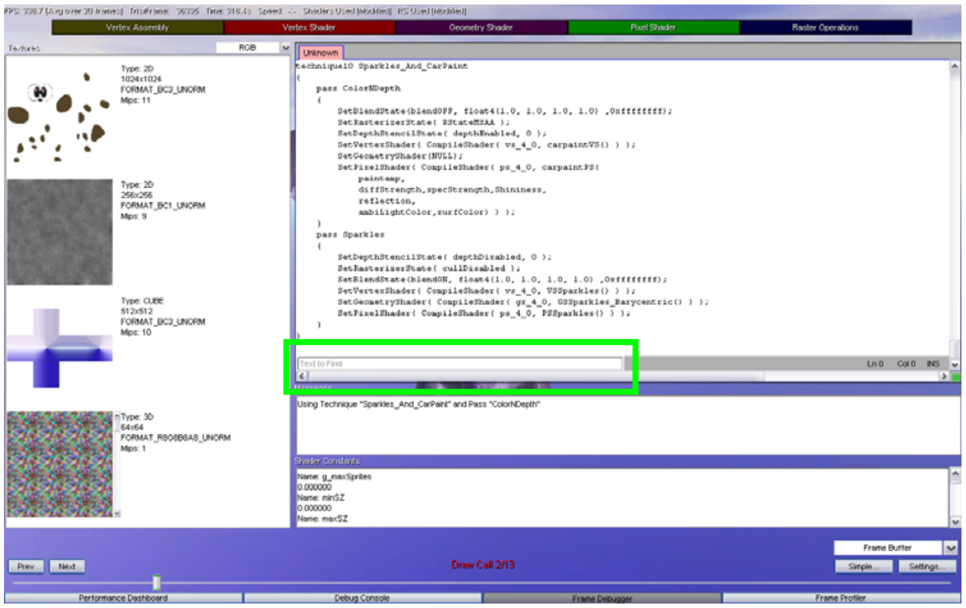
Geometry Shader State Inspector

This state inspector is similar to the Vertex Shader state inspector, showing any geometry shader code, textures, and constants.

Click on the green **Pixel Shader** block at the top of the screen.

Pixel Shader State Inspector

The Pixel Shader state inspector is similar to the Vertex Shader and Geometry Shader state inspectors, showing any geometry shader code, textures, and constants.

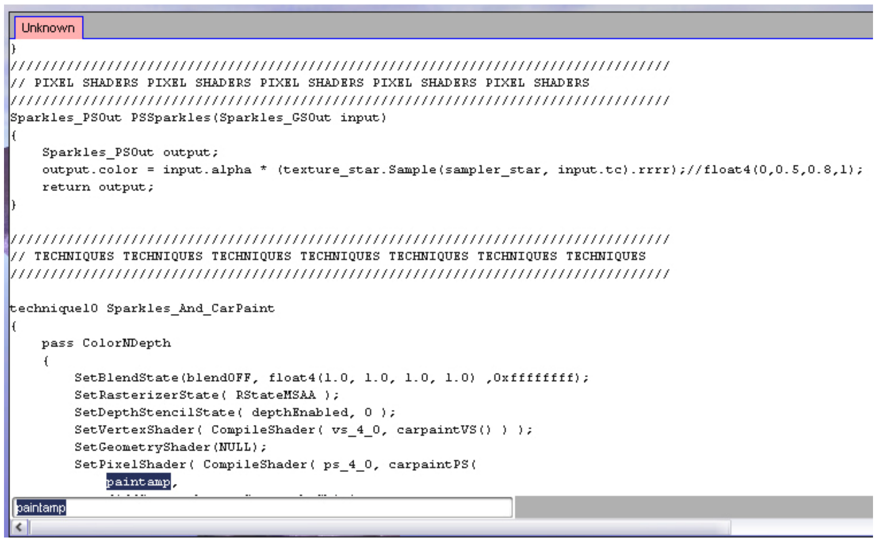


```
technique10 Sparkles_And_CarPaint
{
    pass ColorNDepth
    {
        SetBlendState(blendOFF, float4(1.0, 1.0, 1.0, 1.0) ,0xffffffff);
        SetRasterizerState( RStateMSAA );
        SetDepthStencilState( depthEnabled, 0 );
        SetVertexShader( CompileShader( vs_4_0, carpaintVS() ) );
        SetGeometryShader(NULL);
        SetPixelShader( CompileShader( ps_4_0, carpaintPS(
            paintamp,
            diffStrength,specStrength,Shininess,
            reflection,
            ambiLightColor,surfColor) ) );
    }
    pass Sparkles
    {
        SetDepthStencilState( depthDisabled, 0 );
        SetRasterizerState( cullDisabled );
        SetBlendState(blendON, float4(1.0, 1.0, 1.0, 1.0) ,0xffffffff);
        SetVertexShader( CompileShader( vs_4_0, VSSparkles() ) );
        SetGeometryShader( CompileShader( gs_4_0, GSSparkles_Barycentric() ) );
        SetPixelShader( CompileShader( ps_4_0, PSSparkles() ) );
    }
}
```

```
Using Technique "Sparkles_And_CarPaint" and Pass "ColorNDepth"
```

```
Name: g_maxSprites
0.000000
Name: minSZ
0.000000
Name: maxSZ
```

The search field (shown in green above) allows you to quickly find a particular text string. Type **paintamp** into the search field and press **Enter**. The shader editor will jump to the first occurrence of paintamp.



Unknown

```
}
////////////////////////////////////////////////////////////////////////////////////
// PIXEL SHADERS PIXEL SHADERS PIXEL SHADERS PIXEL SHADERS PIXEL SHADERS
////////////////////////////////////////////////////////////////////////////////////
Sparkles_PSOut PSSparkles(Sparkles_GSOut input)
{
    Sparkles_PSOut output;
    output.color = input.alpha * (texture_star.Sample(sampler_star, input.tc).rrrr);//float4(0,0.5,0.8,1);
    return output;
}

////////////////////////////////////////////////////////////////////////////////////
// TECHNIQUES TECHNIQUES TECHNIQUES TECHNIQUES TECHNIQUES TECHNIQUES TECHNIQUES
////////////////////////////////////////////////////////////////////////////////////

technique10 Sparkles_And_CarPaint
{
    pass ColorNDepth
    {
        SetBlendState(blendOFF, float4(1.0, 1.0, 1.0, 1.0) ,0xffffffff);
        SetRasterizerState( RStateMSAA );
        SetDepthStencilState( depthEnabled, 0 );
        SetVertexShader( CompileShader( vs_4_0, carpaintVS() ) );
        SetGeometryShader(NULL);
        SetPixelShader( CompileShader( ps_4_0, carpaintPS(
            paintamp,
```

paintamp

Now, replace **paintamp** with **0.5**. Then **right-click** in the editing area and choose **Compile** from the context menu. (You can also save and load your shaders using the context menu.)

Your modified shader is now used by the application. Press **F2** to hide PerfHUD's user interface, so you can see the modified rendering.

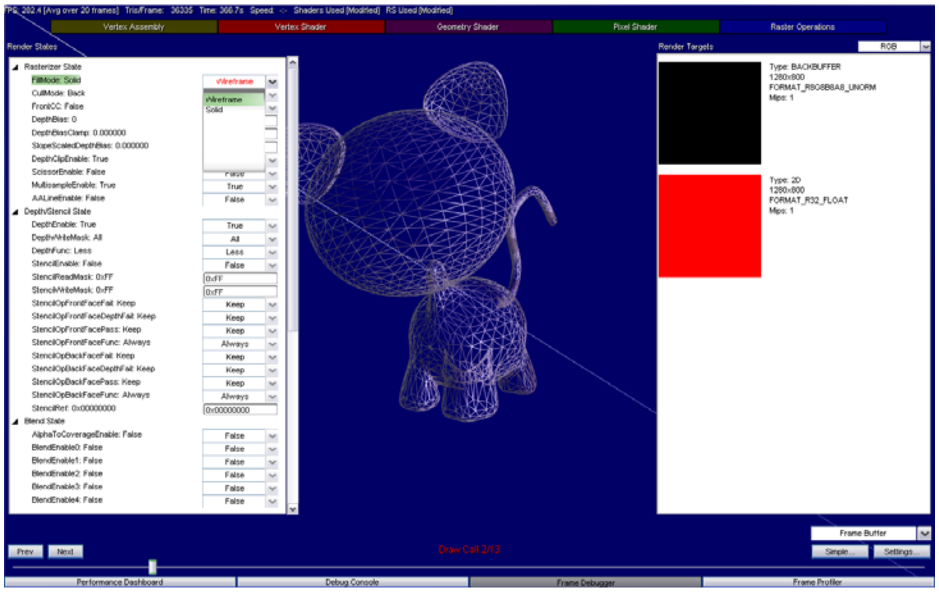
Revert the shader to its original form by **right-clicking** in the editing area and choosing **Revert to Original Shader**.

Next, click on the blue **Raster Operations** block at the top of the screen.

Raster Operations State Inspector

The Raster Operations state inspector allows you to view and manipulate numerous useful render states. Any changes you make here affect all draw calls in the scene. (Future versions of PerfHUD will allow you to affect draw calls grouped by state buckets.)

Select the first dropdown (for the Fillmode) and change it to **Wireframe**. Your screen should now look like this:



Rasterizer State
- FillMode: Solid — Wireframe (Wireframe / Solid)
- CullMode: Back
- FrontCC: False
- DepthBias: 0
- DepthBiasClamp: 0.000000
- SlopeScaledDepthBias: 0.000000
- DepthClipEnable: True
- ScissorEnable: False — False
- MultisampleEnable: True — True
- AALineEnable: False — False

Depth/Stencil State
- DepthEnable: True — True
- DepthWriteMask: All — All
- DepthFunc: Less — Less
- StencilEnable: False — False
- StencilReadMask: 0xFF — 0xFF
- StencilWriteMask: 0xFF — 0xFF
- StencilOpFrontFaceFail: Keep — Keep
- StencilOpFrontFaceDepthFail: Keep — Keep
- StencilOpFrontFacePass: Keep — Keep
- StencilOpFrontFaceFunc: Always — Always
- StencilOpBackFaceFail: Keep — Keep
- StencilOpBackFaceDepthFail: Keep — Keep
- StencilOpBackFacePass: Keep — Keep
- StencilOpBackFaceFunc: Always — Always
- StencilRef: 0x00000000 — 0x00000000

Blend State
- AlphaToCoverageEnable: False — False
- BlendEnable0: False — False
- BlendEnable1: False — False
- BlendEnable2: False — False
- BlendEnable3: False — False
- BlendEnable4: False — False

Type: BACKBUFFER
1280x800
FORMAT_R8G8B8A8_UNORM
Mips: 1

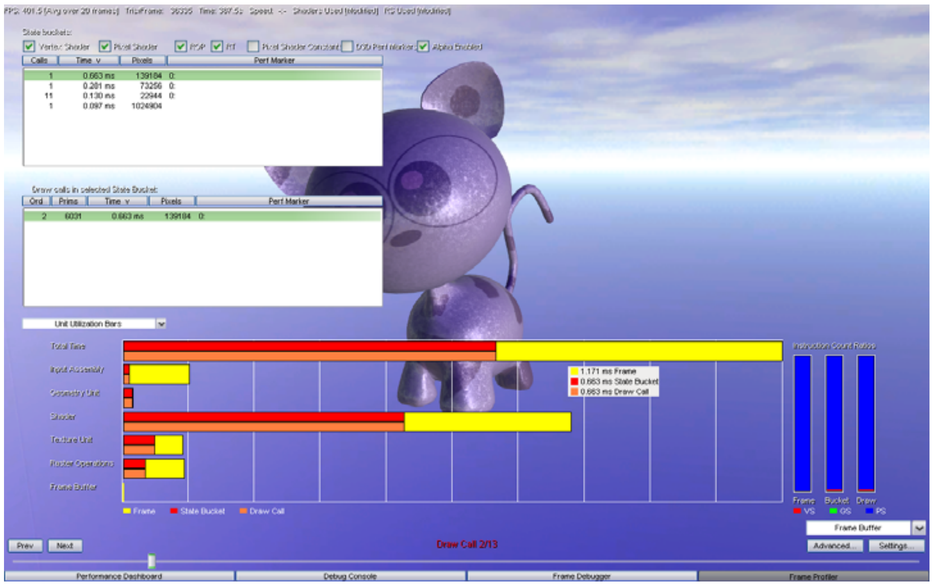
Type: 2D
1280x800
FORMAT_R32_FLOAT
Mips: 1

Now right-click on that same drop-down and select **Restore All States** from the resulting context menu. Note that you can restore states by category if you want to.

Frame Profiler

Press **F8** to enter the Frame Profiler. You'll see PerfHUD quickly run a series of tests on the current frame, giving you detailed statistics about draw call performance and GPU usage. This is one of the uniquely powerful features of PerfHUD – complete bottleneck analysis with just one key press.

The Frame Profiler offers several different visualizations, which are listed and explained briefly below.



[✓] Vertex Shader [✓] Pixel Shader [✓] RSP [✓] RT [ ] Pixel Shader Constant [ ] D3D Perf Marker [✓] Alpha Enabled

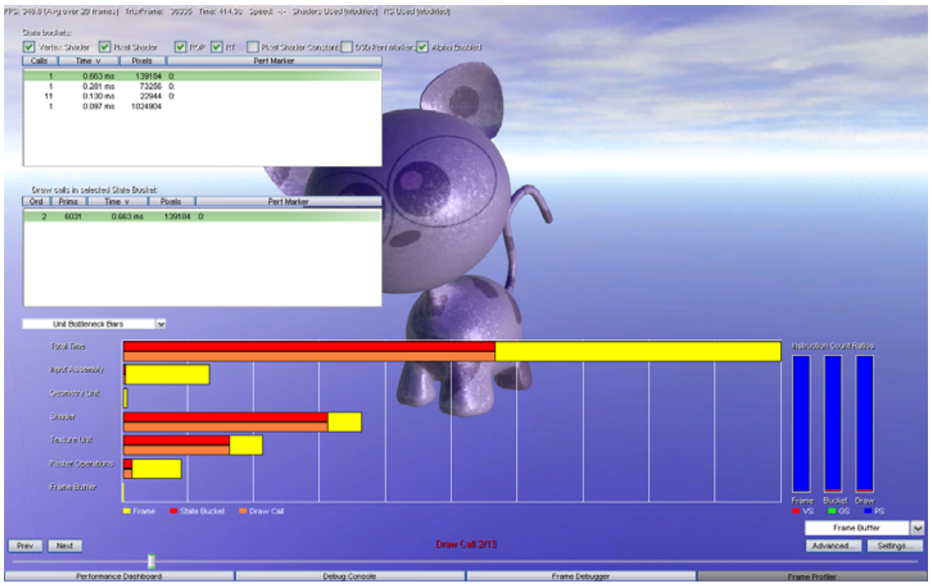
Calls	Time v	Pixels	Perf Marker
1	0.663 ms	139184	0:
1	0.201 ms	73256	0:
11	0.130 ms	22944	0:
1	0.097 ms	1024904	

Draw calls in selected State Bucket:

Ord	Prims	Time v	Pixels	Perf Marker
2	6031	0.663 ms	139184	0:

🟨 1.171 ms Frame
🟥 0.663 ms State Bucket
🟧 0.663 ms Draw Call

Unit Utilization Bars. Shows how long each GPU unit was used for the selected draw call, state bucket, and frame. You can define state bucket groupings using the checkboxes at the top of the screen.



FPS: 348.8 (Avg over 20 frames) Tris/Frame: 36335 Time: 414.3s Speed: -:- Shaders Used (Modified) RS Used (Modified)

State buckets:

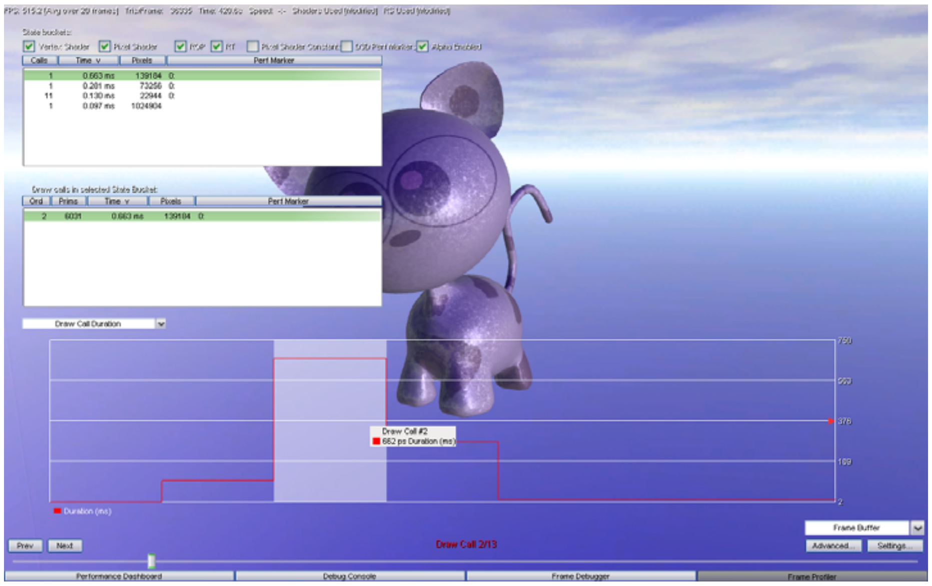
[✓] Vertex Shader [✓] Pixel Shader [✓] ROP [✓] RT [ ] Pixel Shader Constant [ ] D3D Perf Marker [✓] Alpha Enabled

Calls	Time v	Pixels	Perf Marker
1	0.663 ms	139184	0:
1	0.201 ms	73256	0:
11	0.130 ms	22944	0:
1	0.097 ms	1024904	

Draw calls in selected State Bucket:

Ord	Prims	Time v	Pixels	Perf Marker
2	6031	0.663 ms	139184	0:

Unit Bottleneck Bars. Shows how long each GPU unit was the bottleneck for the selected draw call, state bucket, and frame. You can define state bucket groupings using the checkboxes at the top of the screen.



FPS: 515.2 (Avg over 20 frames) Tris/Frame: 36335 Time: 420.6s Speed: -:- Shaders Used (Modified) RS Used (Modified)

State buckets:

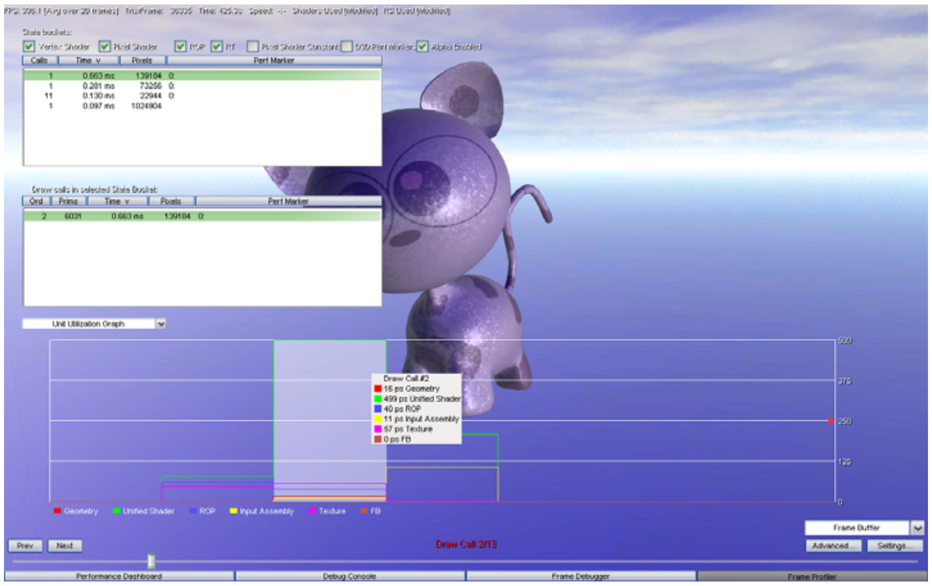
[✓] Vertex Shader [✓] Pixel Shader [✓] ROP [✓] RT [ ] Pixel Shader Constant [ ] D3D Perf Marker [✓] Alpha Enabled

Calls	Time v	Pixels	Perf Marker
1	0.663 ms	139184	0:
1	0.201 ms	73256	0:
11	0.130 ms	22944	0:
1	0.097 ms	1024904	

Draw calls in selected State Bucket:

Ord	Prims	Time v	Pixels	Perf Marker
2	6031	0.663 ms	139184	0:

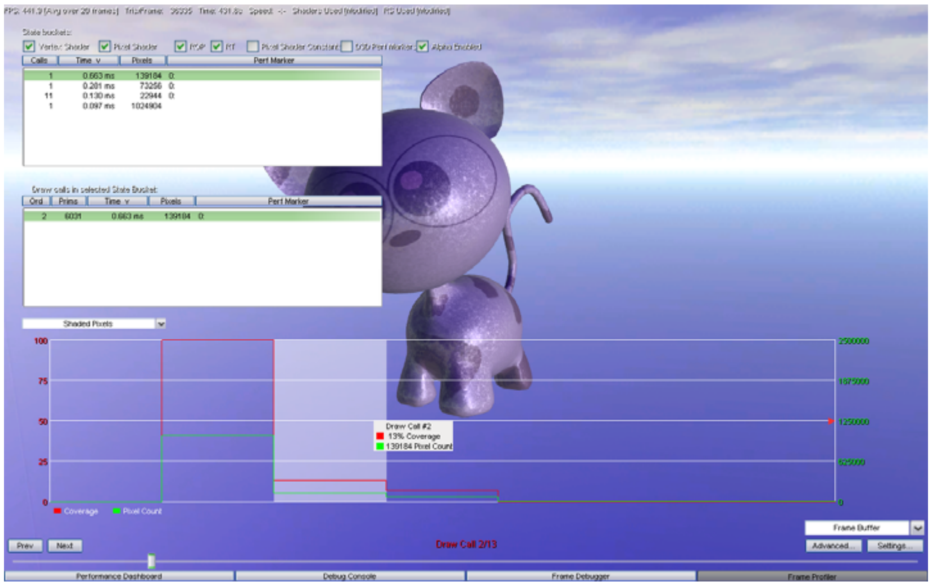
Draw Call Duration. Shows how long each draw call in the frame took. (The horizontal axis is draw call number.) You can click to jump to a draw call, and see tooltips to get exact values.



Calls	Time v	Pixels	Perf Marker
1	0.663 ms	139184	0:
1	0.201 ms	73256	0:
11	0.130 ms	22944	0:
1	0.097 ms	1024904	

Ord	Prims	Time v	Pixels	Perf Marker
2	6031	0.663 ms	139184	0:

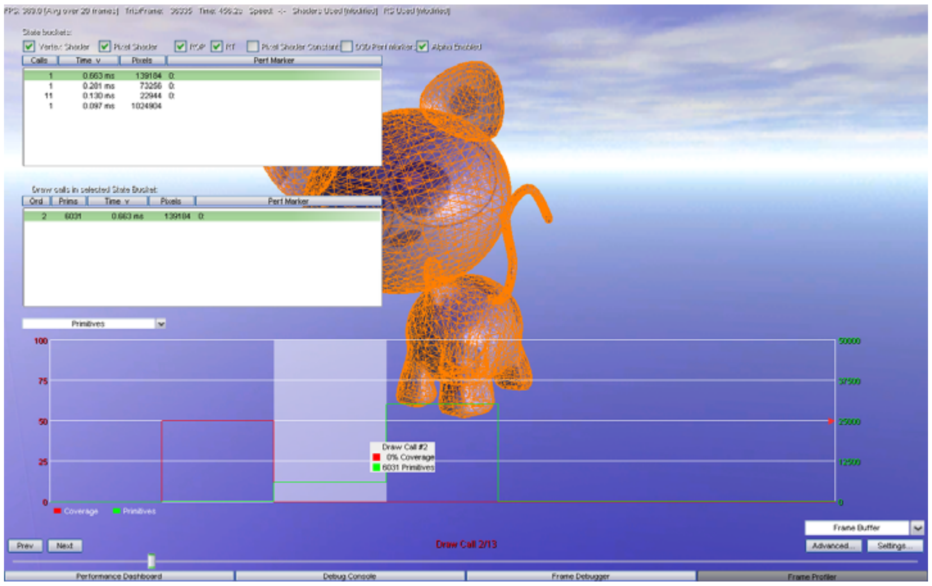
Unit Utilization Graph. Shows how much each GPU unit was utilized for each draw call in the frame. You can click to jump to a draw call, and see tooltips to get exact values.



Calls	Time v	Pixels	Perf Marker
1	0.663 ms	139184	0:
1	0.201 ms	73256	0:
11	0.130 ms	22944	0:
1	0.097 ms	1024904	

Ord	Prims	Time v	Pixels	Perf Marker
2	6031	0.663 ms	139184	0:

Shaded Pixels. Shows how many pixels were drawn by each draw call, as well as what percentage of the screen was covered by the draw call. You can click to jump to a draw call, and see tooltips to get exact values.

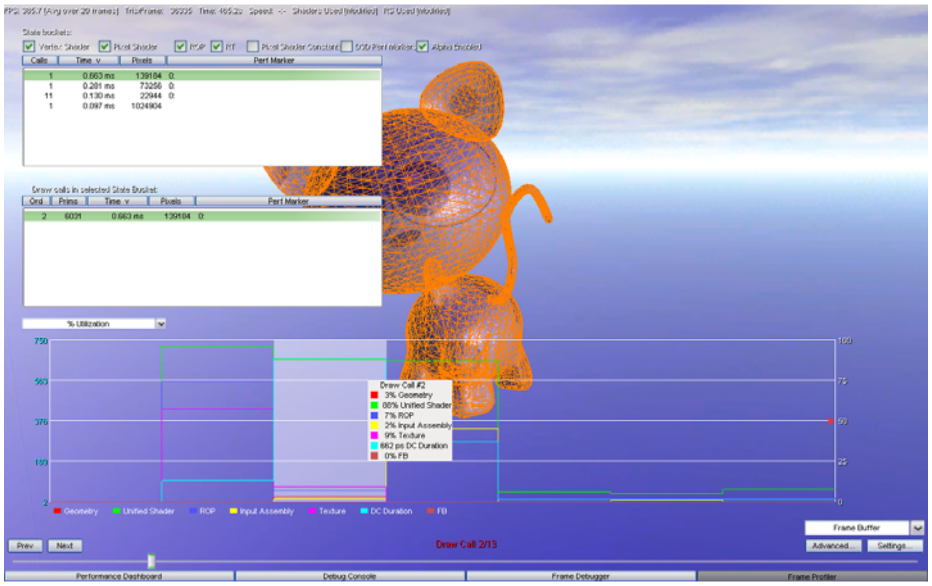


Calls	Time v	Pixels	Perf Marker
1	0.663 ms	139184	0:
1	0.201 ms	73256	0:
11	0.130 ms	22944	0:
1	0.097 ms	1024904	

Ord	Prims	Time v	Pixels	Perf Marker
2	6031	0.663 ms	139184	0:

Draw Call #2
🟥 0% Coverage
🟩 6031 Primitives

Primitives. Shows the number of primitives drawn by each draw call, along with the percentage of the screen covered. You can click to jump to a draw call, and see tooltips to get exact values.



Calls	Time v	Pixels	Perf Marker
1	0.663 ms	139184	0:
1	0.201 ms	73256	0:
11	0.130 ms	22944	0:
1	0.097 ms	1024904	

Ord	Prims	Time v	Pixels	Perf Marker
2	6031	0.663 ms	139184	0:

Draw Call #2
🟥 3% Geometry
🟩 88% Unified Shader
🟦 7% ROP
🟨 2% Input Assembly
🟪 9% Texture
🟦 662 ps DC Duration
🟥 0% FB

% Utilization. Shows how utilized each GPU unit was for each draw call. You can click to jump to a draw call, and see tooltips to get exact values.