W3DUnit

A user-acceptance driven validation tool for Warp3D drivers.

About

W3DUnit is a tool that was written to ensure that a given Warp3D driver behaves in the manner specified by the Warp3D API. This tool was developed in tandem with the Permedia2 driver for OS4 as a means of verifying the proper operation of the supported features. However, it should work with any Warp3D driver.

Purpose

Bugs in hardware-accelerated 3D applications can prove difficult to diagnose due to number of layers between the the application and the rasterisation. A typical OS4 native application that uses 3D will usually use MiniGL, which in turn uses the Warp3D.library which finally uses a hardware specific driver for rendering. Each of these layers may have bugs that are only visible under certain circumstances. For the same reason, bugs present in drivers are difficult to diagnose by simply testing a set of known 3D applications since it is very unlikely that a sufficiently broad section of a given driver's functionality will be tested.

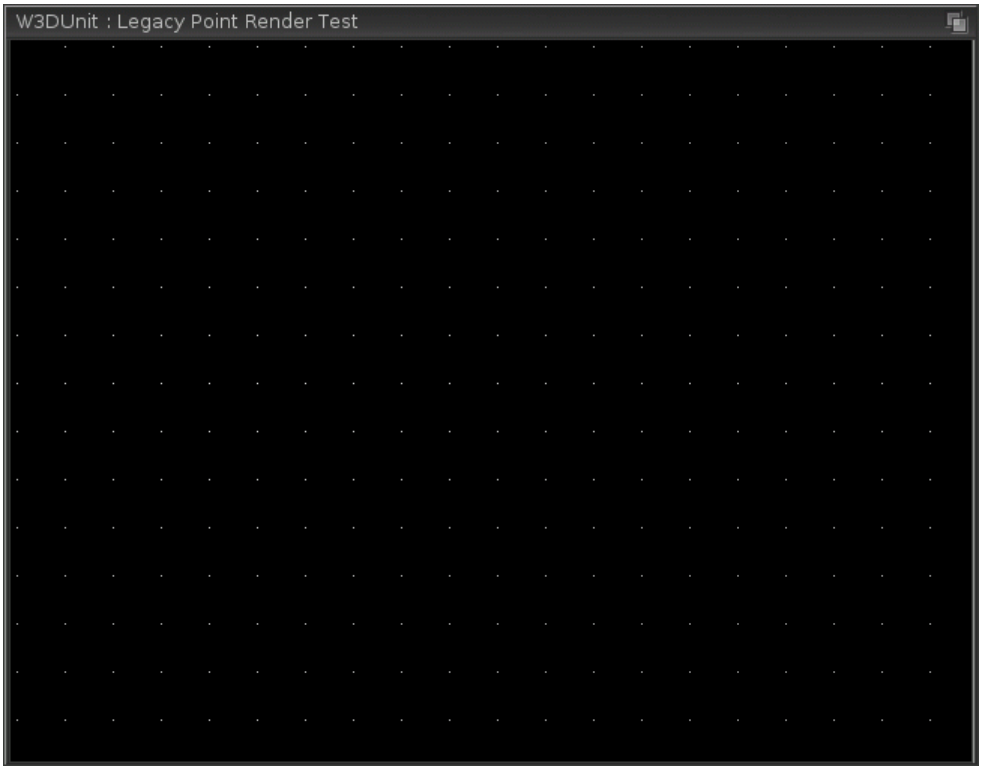
W3DUnit addresses this issue by opening Warp3D directly and invoking it's API calls in a series of controlled tests that result in simple to interpret pass/fail cases.

2 Legacy Drawing API Tests

The original Warp3D API provided calls for the rendering of points, lines, triangles, triangle strips and triangle fans that took discrete structures for each class of primitive as parameters. These structures in turn directly embedded fixed-format vertex data. Although very simple, the scheme was not very flexible. In V3 of the API, vectorised versions of these calls and their corresponding data strictures were added for triangles, fans and strips that at least separated the structures used to represent the primitives from their corresponding vertex data.

2.1 Legacy Point Render Test

This tests the rendering of points using the W3D_DrawPoint() function. Most drivers will include this function for completeness but it will almost always be slower than rendering points in software. If the operation is properly supported, a grid of evenly spaced light grey pixels should be rendered as per the illustration below.



Possible problems that may be encountered with the W3D_DrawPoint() function test are:

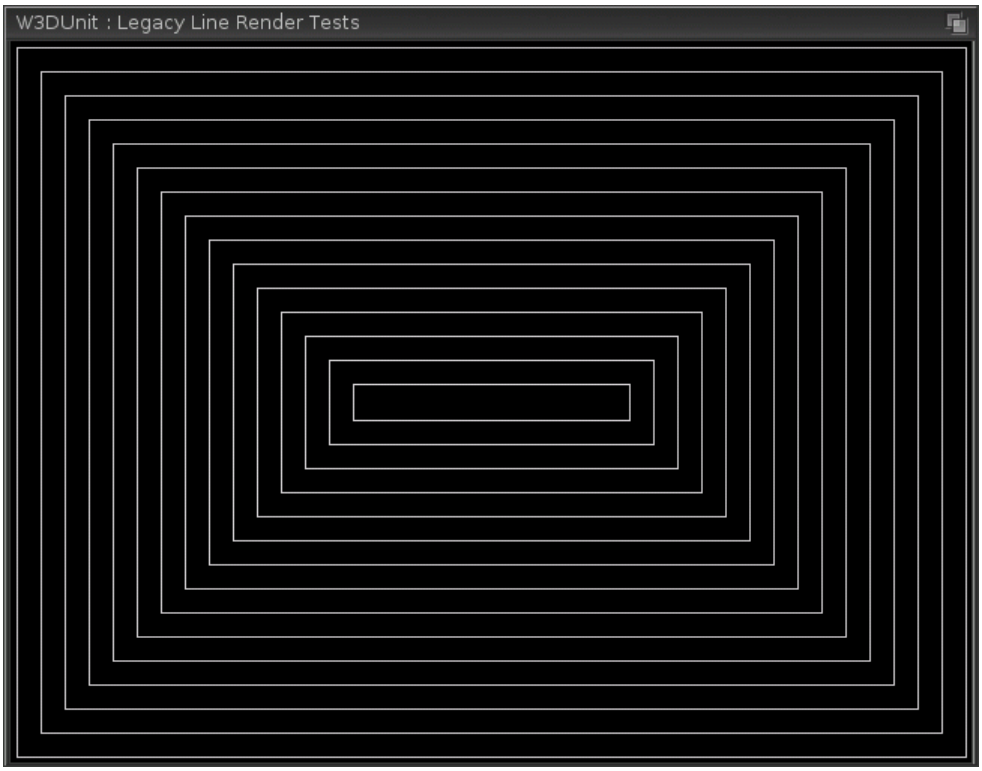
- Nothing rendered. This suggests that the function may not be implemented.
- Points are rendered but with possible missing rows or columns in the grid. This suggests the driver may have issues with the definition of pixel origin where sub-pixel precision is used.

2.2 Legacy Line Render Test

This tests the rendering of lines using the following family of functions:

- W3D_DrawLine(). Four calls per rectangle, 8 vertices total.
- W3D_DrawLineStrip(). One call per rectangle, 5 vertices total.
- W3D_DrawLineLoop(). One call per rectangle, 4 vertices total.

For each function, a series of nested, evenly-spaced wireframe light-grey rectangles should be rendered as per the figure below.



Possible problems with the W3D_DrawLine() function test are:

- Nothing rendered. This suggests that the function may not be implemented.

Possible problems with the W3D_DrawLineStrip() function test are:

- Nothing rendered. This suggests that the function may not be implemented.
- Unexpected diagonals. This suggests a problem in the basic vertex set up for line rendering.

Possible problems with the W3D_DrawLineLoop() function test are:

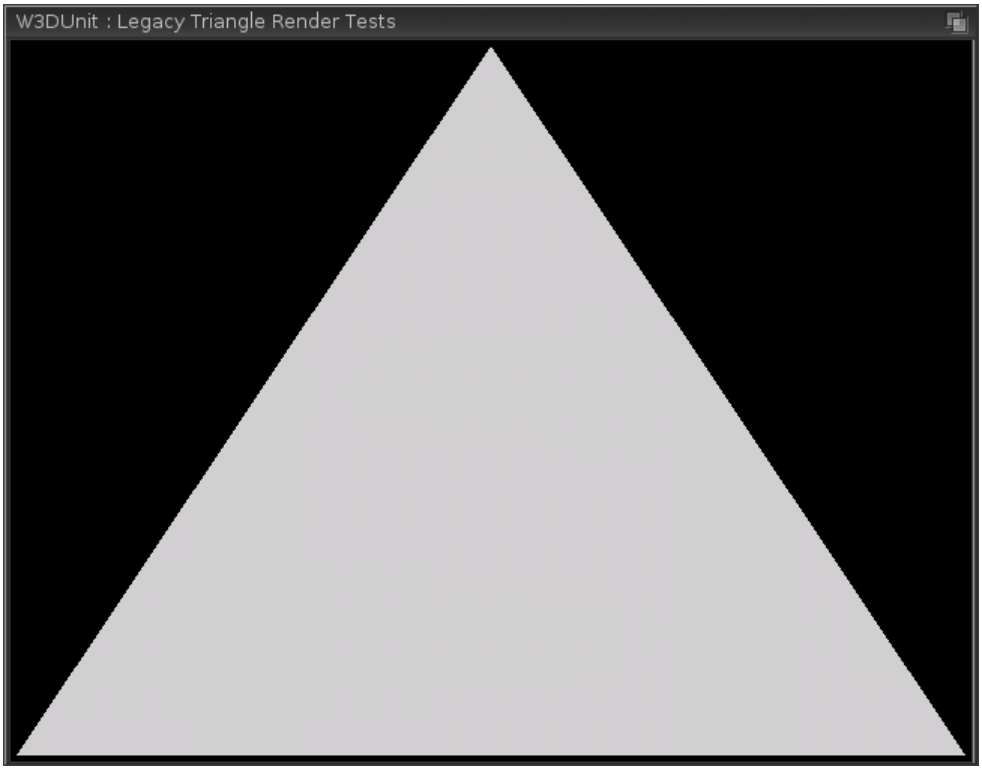
- Nothing rendered. This suggests that the function may not be implemented.
- Unexpected diagonals. This suggests a problem in the basic vertex set up for line rendering.
- Left-hand side of rectangles missing. This suggests a problem in the line-closing termination of the loop.

2.3 Legacy Triangle Test

This tests the rendering of single triangles using the following functions:

- W3D_DrawTriangle(). Accepts W3D_Triangle structure embedding 3 W3D_Vertex instances.
- W3D_DrawTriangleV(). Accepts W3D_TriangleV structure embedding 3 W3D_Vertex pointers.

For each function, a single, solid light grey triangle should be rendered as per the illustration below.



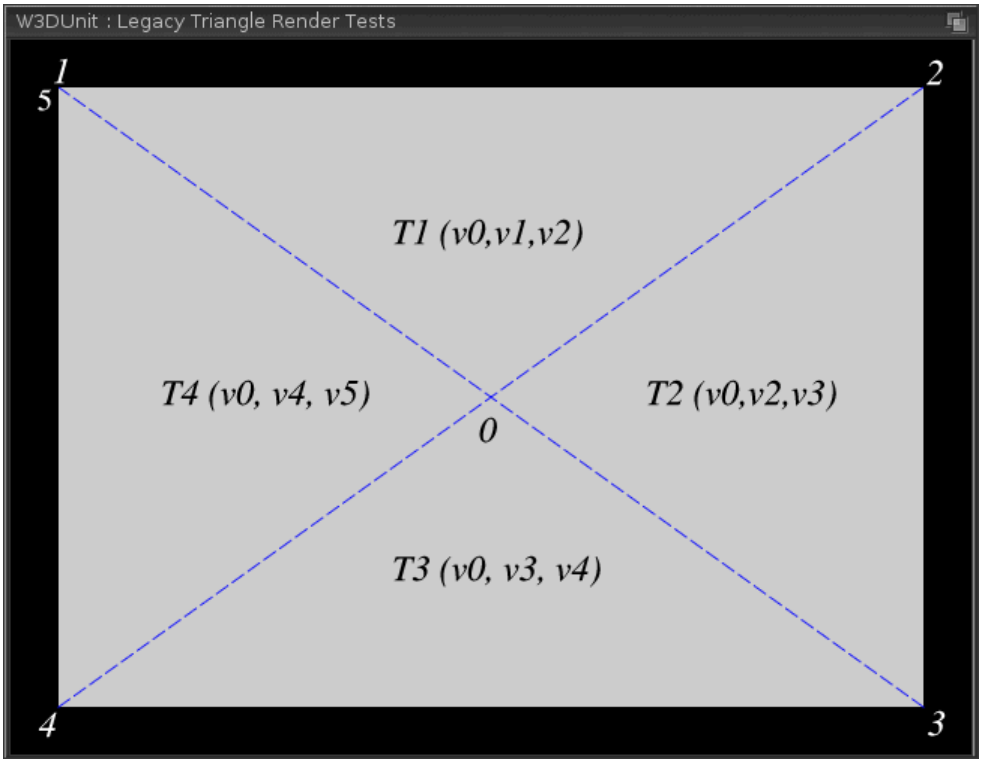
For all intents and purposes, both functions are identical. For production drivers, no problems are to be expected with either call since these functions are considered fundamental to the API.

2.4 Legacy Triangle Fan Test

This tests the rendering of triangle fans using the following functions:

- W3D_DrawTriFan()
- W3D_DrawTriFanV()

For each function, a solid light grey 5-vertex triangle fan is rendered that has vertex 0 located at the centre and the remaining four arranged in the corners of a rectangle, which should result in a simple solid grey rectangle as illustrated below. Note that the dashed lines and text serve to illustrate the triangle and vertex locations for this documentation but are not part of the render output.



As with the triangle rendering tests above, these functions are considered fundamental and should not have problems. However, most uses of these functions are for rendering quads, and as such, higher vertex-count fans may not have been fully tested. Hence, possible problems with either function test are:

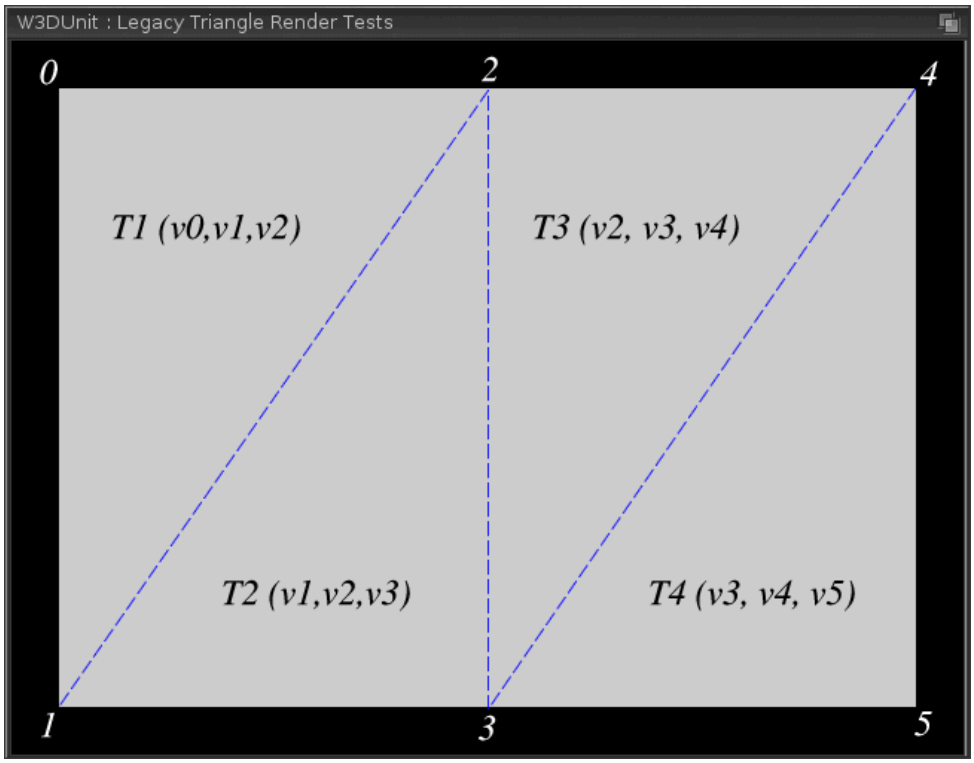
- Triangular segment(s) missing. This suggests the fan rendering loop may have an early exit bug.
- Malformed rectangle. This suggests the fan rendering loop may be only using edge, rather than single common vertex and edge sharing, resulting in strip, rather than fan behaviour for larger vertex counts.

2.5 Legacy Triangle Strip Test

This tests the rendering of triangle strips using the following functions:

- W3D_DrawTriStrip()
- W3D_DrawTriStripV()

For each function, a solid light grey 6-vertex strip is rendered to produce a regular 4-triangle strip with the same overall render appearance of the triangle fan. The topology of the strip is illustrated below.

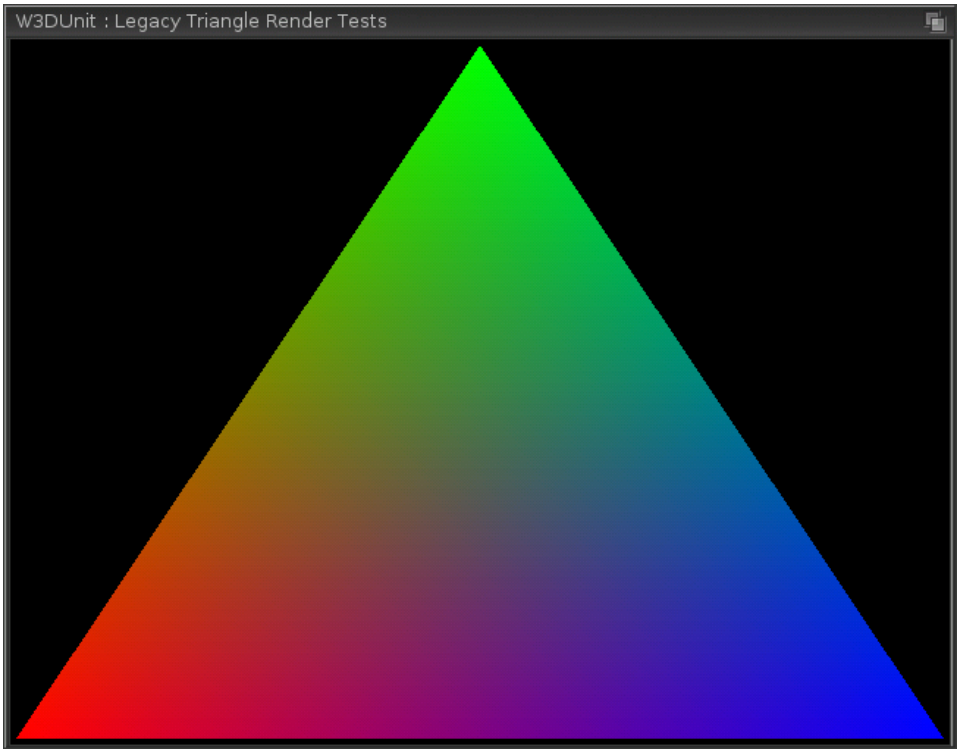


As with the triangle rendering tests, these functions are considered fundamental and should not have problems. However, most uses of these functions are for rendering quads, and as such, higher vertex-count strips may not have been fully tested. Hence, possible problems with either function test are:

- Triangular segment(s) missing. This suggests the strip rendering loop may have an early exit bug.
- Malformed rectangle. This suggests the strip rendering loop may be a single common vertex as well as edge sharing, rather than edge sharing alone, resulting in fan, rather than strip behaviour for larger vertex counts. For example, if vertex 0 were shared as in the triangle fan arrangement, the bottom edges of T2 and T3 would be missing and instead be replaced by slopes from v0 to v3, and v0 to v5 respectively, producing a somewhat sawtooth like appearance overall.

2.6 Shading Test

This tests checks that Gouraud shading, another fundamental feature, is properly implemented. In the legacy primitive tests, all vertex colours were set the same as the flat shade colour since any shading bugs that might exist could detract from the fundamental operation being tested. In order to ensure that shading does work, the shading test reuses the same triangle as seen in the legacy triangle test with the bottom left vertex set to red, the top centre vertex set to green and the bottom right vertex set to blue. It then renders the triangle with shading enabled and then a second time disabled. If Gouraud shading is properly implemented, the triangle should be rendered rather more colourfully, as illustrated below.



When Gouraud shading is disabled, the triangle should be rendered again in solid light grey.

As with triangle rendering generally, support for shading is considered fundamental and bugs are not to be expected in production drivers. However, depending on the hardware, flat shading might not be genuinely supported and faked by setting all vertices the same colour. This can potentially lead to problems if not handled correctly.

Possible problems encountered with this test are:

- When switching to flat shading, vertex colours remain unchanged. This suggests a state-handling bug in the driver or a false assumption that vertex colours do not need to be (re)set to achieve flat shading.

2.7 Z Buffer Test

This tests that depth comparison using a Z buffer, another fundamental feature, is properly implemented. The Z buffer is simply a buffer that stores the computed z- value of rendered pixels to some hardware-defined precision which can in turn be used to test whether or not part of a primitive is occluded by a previously rendered one using a comparison function from the list below.

- W3D_Z_NEVER. Never draw incoming pixel.
- W3D_Z_LESS. Draw incoming pixel if Z(pixel) < Z(buffer) .
- W3D_Z_LEQUAL. Draw incoming pixel if Z(pixel) <= Z(buffer).
- W3D_Z_EQUAL. Draw incoming pixel if Z(pixel) = Z(buffer) .
- W3D_Z_GEQUAL. Draw incoming pixel if Z(pixel) >= Z(buffer) .
- W3D_Z_GREATER. Draw incoming pixel if Z(pixel) > Z(buffer) .
- W3D_Z_NOTEQUAL. Draw incoming pixel if Z(pixel) <> Z(buffer) .
- W3D_Z_ALWAYS. Always draw incoming pixel.

The Warp3D API allows the Z buffer to be switched between read/write and read-only for the purposes. The Z buffer test is split into two parts. The first part tests that all greater than / less than comparisons in the above list work. The second part tests that all equality comparisons in the above list work. The reason that the test is split into two parts is that it is difficult to test comparison functions like W3D_GEQUAL and differentiate the results from W3D_GREATER without knowing the specific hardware precision of the Z buffer in advance.

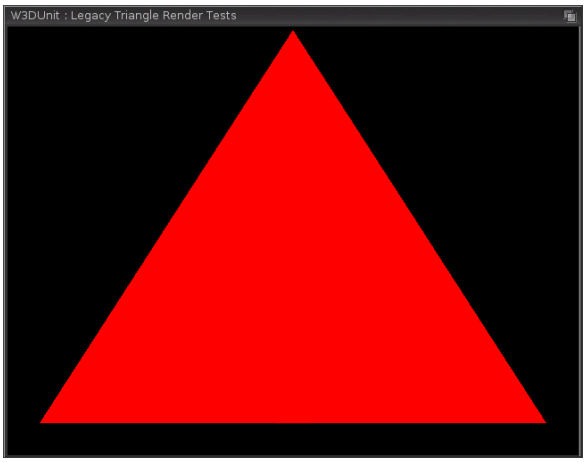
In the first part of the test, two overlapping triangles are set up, one red and one blue. The red triangle, which points upwards, is configured such that the horizontal base has a Z value close to 1 (“far” behind the screen), and the vertex at the top has a Z value close to 0 (“close” to the screen). The blue triangle, which points downwards is configured in the same fashion such that both triangles naturally share an intersection along the horizontal line where Z is 0.5 for each triangle. This arrangement makes it easy to interpret the output of simple inequalities such as W3D_Z_LESS. In each case, the background Z value is initially cleared to something appropriate such that both triangles are always rendered.

In the second part of the test, the two triangles are each reconfigured such that their vertex Z values are 0.5 and are thus completely coplanar, which makes it easy to interpret the output of straight equality/inequality cases such as W3D_Z_EQUAL and W3D_Z_NOT_EQUAL. Cases such as W3D_Z_LEQUAL must satisfy both parts of the test to be considered as correctly implemented. As for the first part of the test, the background Z value is initially cleared to something appropriate such that the both triangles should be fully rendered. An exception to this is in the W3D_Z_EQUAL test where the background value is deliberately different such that it can be disambiguated from similar cases.

The Z buffer tests will only be performed if a Z buffer could be allocated. In total, there are ten discrete tests, the desired outcomes of which are described below.

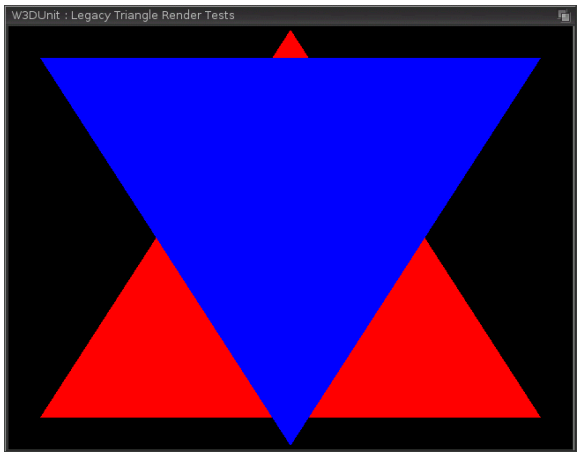
Part 1: W3D_Z_NEVER

In the W3D_NEVER test, only the red triangle should be rendered, as illustrated below.



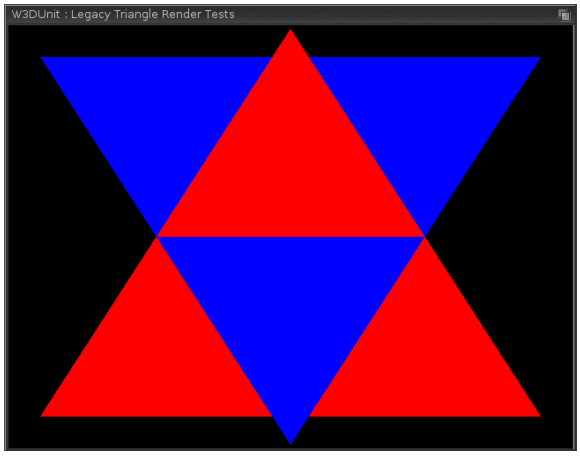
Part 1: W3D_Z_ALWAYS

In this test, both triangles should be rendered with every part of the blue triangle rendered such that it appears to be in front of the red triangle, as illustrated below.



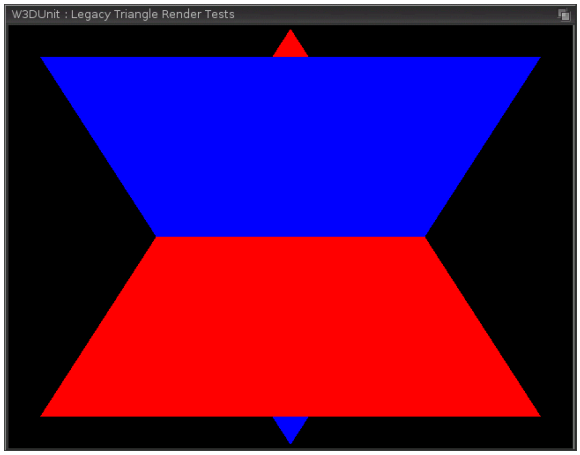
Part 1: W3D_Z_LESS, W3D_Z_LEQUAL

In these tests, the red and blue triangles should intersect such that their tips appear to be in front of the horizontal base of their counterpart, as illustrated below.



Part 1: W3D_Z_GREATER, W3D_Z_GEQUAL

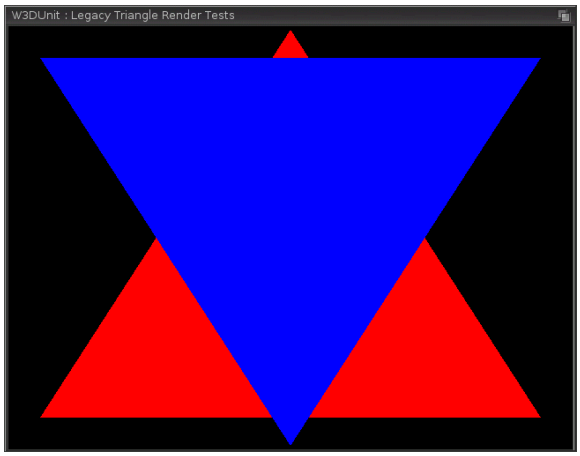
In these tests, the red and blue triangles should intersect such their horizontal bases appear to be in front of the tip of their counterpart, as illustrated below.



This concludes the part1 tests. After this stage, four more tests are performed in order to test the W3D_Z_EQUAL, W3D_Z_NOTEQUAL and fully validate the W3D_Z_LEQUAL and W3D_Z_GEQUAL comparison functions. In these remaining tests, the triangles are reconfigured to be coplanar, thus having identical depths in the Z buffer.

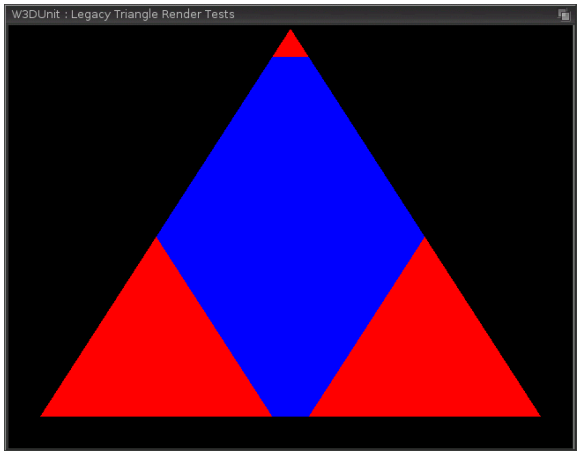
Part 2: W3D_LEQUAL, W3D_GEQUAL

In these tests, the triangles are coplanar such that only the equality of their depth is tested. The background Z value is cleared to an appropriate value such that both triangles are either less than or greater than (as required) to ensure they render. As such, both the blue and red triangles should render fully and as such appear to be the same as the part 1 W3D_Z_ALWAYS result.



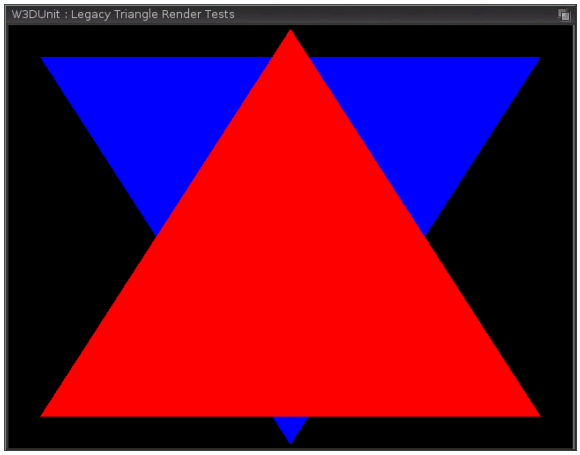
Part 2: W3D_EQUAL

In this test, the triangles are coplanar and only the overlapping area of the blue triangle should be rendered since the only area where the blue triangle's depth matches the existing Z buffer is where the red triangle was already rendered, producing the effect illustrated below.



Part 2: W3D_NOTEQUAL

In this test, the triangles are coplanar and only the non-overlapping area of the blue triangle should be rendered since the only place where the blue triangle's depth matches the existing Z buffer is where the red triangle was already rendered. Consequently, the blue triangle should appear to be entirely behind the red triangle, as illustrated below.



This concludes the part 2 tests. All ten tests should render exactly as described in order for the driver's Z buffer functionality to be considered fully conforming to the Warp3D specification.

Possible problems encountered with this test are:

- Part 2 tests for W3D_Z_LEQUAL / W3D_Z_GEQUAL do not pass. Some hardware may not implement the required logic to properly support these comparisons. However, in the vast majority of real use cases, this would not be a problem since their render output would be virtually indistinguishable from W3D_Z_LESS / W3D_Z_GREATER.

2.8 Vertex Array Geometry Format Test

Version 4 of Warp3D introduced the concept of vertex arrays as a more efficient and flexible approach to rendering primitives than the existing discrete drawing functions. Under this model, the client application can set up arrays of geometry, colour and texture coordinate data in a variety of different formats. For vertex geometry, the client can specify array data in one of three possible formats.

- W3D_VERTEX_F_F_F. Three consecutive floats (32-bit) for X, Y and Z.
- W3D_VERTEX_F_F_D. Two consecutive floats for X and Y, double (64-bit) Z.
- W3D_VERTEX_D_D_D. Three consecutive doubles for X, Y and Z.

Vertex array data is set up in order to render a solid light grey triangle for each of the above formats. In each case, the triangle will be rendered with a call to W3D_DrawArray() after which, no perceptible difference should be observed from the triangle illustrated in part 2.3 should be observed.

Possible problems encountered with this test are:

- Malformed or missing triangle. This suggests that the specific vertex geometry format has not been implemented, the pointer not correctly configured or has possible data-access alignment issues.

2.9 Vertex Colour Format Test

For vertex array data, Warp3D allows five distinct colour formats, each of which can be provided as floating point data in the range 0.0 – 1.0 or unsigned bytes in the range 0 – 255, giving a total of ten possible vertex colour formats.

- W3D_CMODE_RGB|W3D_COLOR_UBYTE
- W3D_CMODE_BGR|W3D_COLOR_UBYTE
- W3D_CMODE_RGBA|W3D_COLOR_UBYTE
- W3D_CMODE_ARGB|W3D_COLOR_UBYTE
- W3D_CMODE_BGRA|W3D_COLOR_UBYTE
- W3D_CMODE_RGB|W3D_COLOR_FLOAT
- W3D_CMODE_BGR|W3D_COLOR_FLOAT
- W3D_CMODE_RGBA|W3D_COLOR_FLOAT
- W3D_CMODE_ARGB|W3D_COLOR_FLOAT
- W3D_CMODE_BGRA|W3D_COLOR_FLOAT

Vertex array data is set up in order to render a triangle in which the bottom left vertex is red, the top centre vertex green and the bottom right vertex blue, with the alpha value for each vertex colour (where applicable) set for maximum opacity. The triangle is then rendered with a call to W3D_DrawArray(), after which no perceptible difference to that illustrated in 2.6 should be observed.

Possible problems encountered with this test are:

- Vertices have arbitrary colours/opacity. This suggests the vertex colour format has not been implemented, colour pointer has not been correctly configured or has data-access alignment issues.
- Vertices are red, green and blue but not in the specified order of bottom-left, top-centre and bottom-right respectively, especially transposition of red and blue. This suggests that the conversion from vertex colour format to device colour format has potential endian issues.
- One or more vertices is magenta, yellow or cyan. This suggests a possible transposition of the vertex colour alpha component with one of the other primary colours. This suggests that the conversion from vertex colour format to device colour format has potential endian issues.

Vertex Array Render Tests

Once the vertex array geometry and colour formats have been validated, rendering using the V4 vertex array methods will be tested. Warp3D provides two methods for vertex array drawing, listed below.

- W3D_DrawArray()
- W3D_DrawElements()

Each function takes an argument that describes the primitive that will be rendered. Supported primitives are listed below.

- W3D_PRIMITIVE_POINTS. Discrete points, *numPoints* vertices are read
- W3D_PRIMITIVE_LINES. Discrete lines, *numLines**2 vertices are read.
- W3D_PRIMITIVE_LINESTRIP. Strip of joined lines, *numLines*+1 vertices are read.
- W3D_PRIMITIVE_LINELOOP. Closed loop of joined lines, *numLines* vertices are read.
- W3D_PRIMITIVE_TRIANGLES. Discrete triangles, *numTris**3 vertices are read.
- W3D_PRIMITIVE_TRIFAN. Fan of triangles, *numTris*+2 vertices are read.
- W3D_PRIMITIVE_TRISTRIP. Strip of triangles, *numTris*+2 vertices are read.

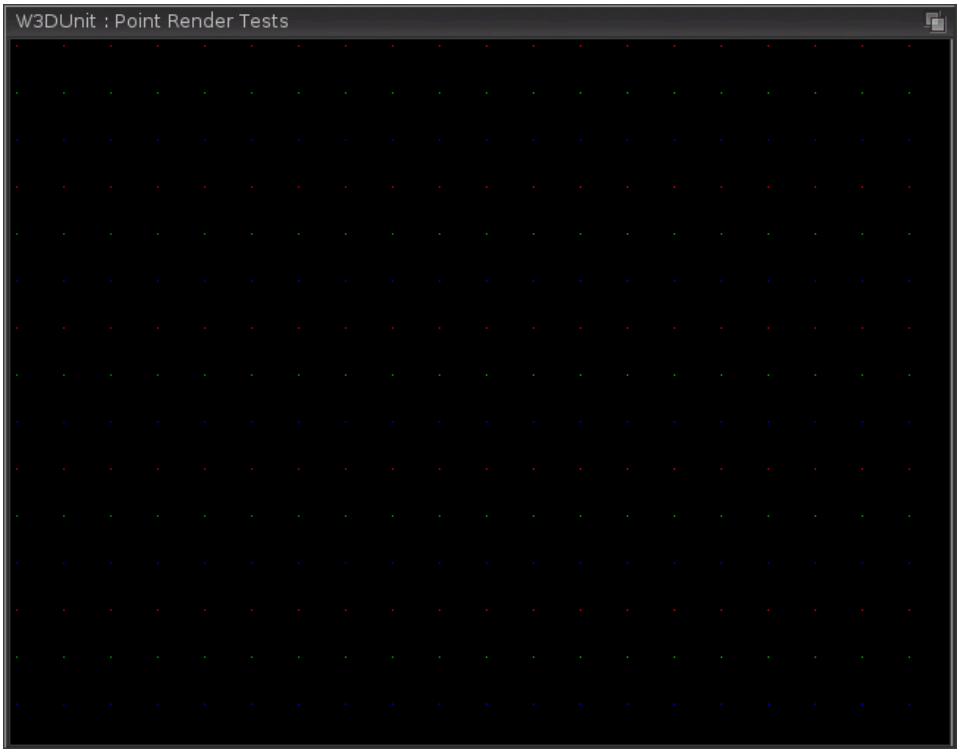
The first function reads vertex data for primitives from client provided arrays sequentially. The second function takes an array of array indices which is read sequentially, and the corresponding vertex array position looked up in the array to retrieve the data. Although not quite equivalent, these two methods can be viewed as the V4 counterparts of the V3 DrawSomething() / DrawSomethingV() pair. For the W3D_DrawElements() function, the format of the index array can be specified as one of the following types.

- W3D_INDEX_UBYTE
- W3D_INDEX_UWORD
- W3D_INDEX_ULONG

For ease of verification, during the following tests, primitives rendered using the W3D_DrawArray() call will be rendered in the same light grey as previous tests. Primitives rendered using the W3D_DrawElements() call will be rendered blue, green or red depending on their index type, with red signifying W3D_INDEX_ULONG, green W3D_INDEX_UWORD and blue W3D_INDEX_UBYTE respetively.

Vertex Array Point Render Tests

This tests the rendering of points via W3D_PRIMITIVE_POINTS, first using W3D_DrawArray() then using W3D_DrawElements(). The output of the W3D_DrawArray() should be a grid of light grey pixels, exactly the same as that illustrated in section 2.1. The output of the W3D_DrawElements() should be the same grid, with alternating rows of red, green and blue pixels, for each of the supported index types respectively, as illustrated below.



Possible problems encountered with these tests are:

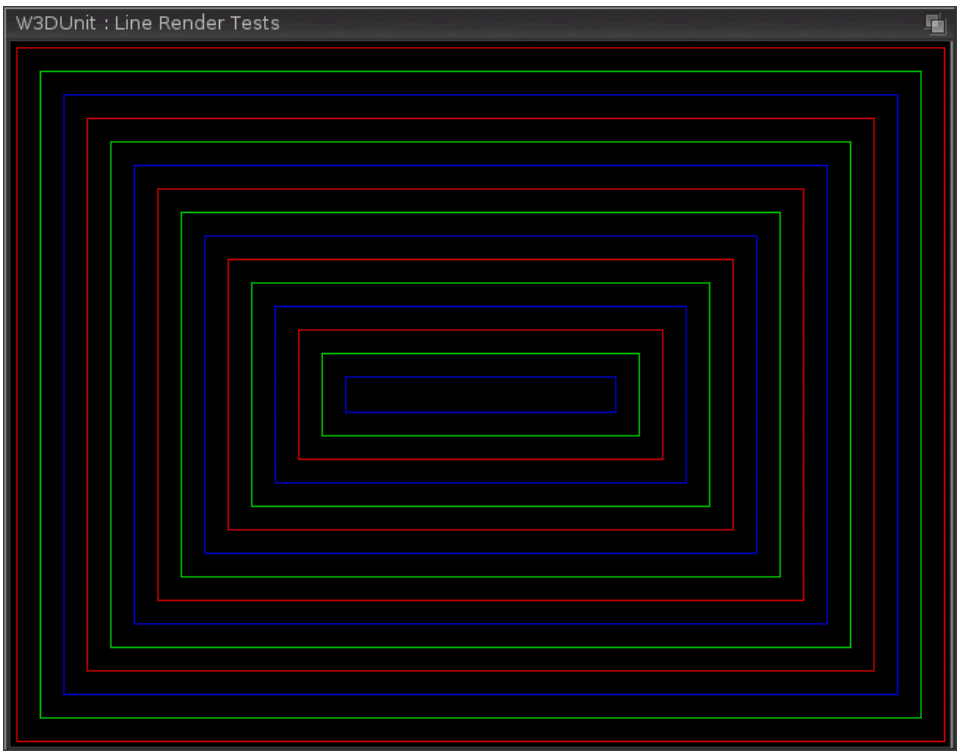
- No points are rendered in either test. This suggests that array support for this primitive has not been implemented.
- Points are rendered for W3D_DrawArray() only. This suggests that the W3D_DrawElements() call is missing the implementation required to support this primitive.
- Points of one or two of the three colours in W3D_DrawElements() are not rendered. This suggests that the W3D_DrawElements() call does not support the corresponding index type for this primitive, or has data access alignment issues.

Vertex Array Line Render Tests

This tests the rendering of lines, line strips and line loops using W3D_DrawArray() and W3D_DrawElements() for the following primitive types:

- W3D_PRIMITIVE_LINES
- W3D_PRIMITIVE_LINESTRIP
- W3D_LINELOOP

For each test, the same configuration as the test described in section 2.2 is used. Each primitive is first rendered using W3D_DrawArray() and then W3D_DrawElements(). The output of each call to W3D_DrawArray() should be a set of nested light grey wireframe rectangles, exactly as illustrated in section 2.2. The output of W3D_DrawElements() should be exactly the same set of rectangles, with colours alternating between red, green and blue for each of the supported index types, as illustrated below.



Possible problems encountered with these tests are:

- No lines are rendered in any given test. This suggests that array support for the particular primitive has not been implemented.
- Lines are rendered for W3D_DrawArray() only. This suggests that the W3D_DrawElements() call is missing the implementation required to support the primitive being tested.
- Lines of one or two of the three colours in W3D_DrawElements() are not rendered. This suggests that the W3D_DrawElements() call does not support the corresponding index type for this primitive, or has data access alignment issues.

Vertex Array Triangle Render Tests

Vertex Array Triangle Fan Render Tests

Vertex Array Triangle Strip Render Tests