Opengl es 2.0 specification pdf

The OpenGL ES register contains specifications of the core API and shadow language; specifications of khronos and supplier-approved OpenGL ES extensions; header files corresponding to the specifications; related documentation. The OpenGL ES register is part of the combined OpenGL register for OpenGL, OpenGL ES and OpenGL SC, which includes the XML API register of reserved listers and functions. Table of Contents OpenGL ES Core API and Shading Language Specifications and Reference Pages The current version of OpenGL ES is OpenGL ES 3.2. Specifications for older versions 3.1, 3.0, 2.0, 1.1 and 1.0 are also available below. See the developer Khronos.org pages for additional specifications, headers, and documentation that are not listed below. Header files that don't have a review date include the last update time in comments at the top of the file. OpenGL ES 3.2 Specifications and documentation OpenGL ES 3.1 Specifications and documentation OpenGL ES 3.0 Specifications and documentation OpenGL ES 2.0 Specifications and documentation OpenGL ES 1.1 Specifications and documentation OpenGL ES 1.0 Specification and Documentation API and Expansion Header Files Because extensions vary from platform to platform and driver to driver, OpenGL ES separates headers for each API version into a header for the core API (OpenGL ES 1.0 , 1.1, 2.0, 3.0, 3.1 and 3.2) and define a separate header extension interfaces for that core API. These header files are delivered here for developers and platform vendors. They define interfaces, including reseners, prototypes, and for platforms that support dynamic runtime extension queies, such as Linux and Microsoft Windows, feature pointer types. Report problems as problems in the OpenGL registry repository. In addition to the main API and expansion headers, there is also an OpenGL ES version-specific platform header file that is designed to define call conventions and data types specific to a platform. Almost all of the headers below depend on a platform header file common to multiple Khronos APIs called <KHR hrplatform.h=>. Vendors can include custom versions of one or all of these headers with their OpenGL ES deployments, but in general, only the platform-specific OpenGL ES and Khronos headers are likely to be modified by the deployment. This makes it possible for developers to drop in more recently updated versions of the headers obtained here, usually when new extensions are delivered on a platform. OpenGL ES 3.2 Headers <GLES3 l32.h=>OpenGL ES 3.2 Header file. <GLES2 l2ext.h=>OpenGL ES Extension Header File (this header is defined to contain all defined extension interfaces for OpenGL ES 2.0 and all later versions, as later versions are backwardcompatible with OpenGL ES 2.0). l3platform.h=>OpenGL ES 3.2 Platform-dependent macros (this header is shared with OpenGL ES 3.0 and 3.1). OpenGL ES 3.1 Headers OpenGL ES 3.0 Headers OpenGL ES 2.0 Headers OpenGL ES 1.1 Headers Khronos</GLES3> </GLES2> </GLES3> Platform Header (<KHR hrplatform.h=>) Expansion specifications by number Open topic with navigation data-mc-breadcrumbs-count=3 data-mc-toc=True>You are here: Expansion support in EGL and GLES differs on a device-by-device and OS-by-OS basis on Tegra. Applications should always ask for support for extensions on the target platform. Some of the Khronos general GLES extensions supported on most Tegra platforms are the following. The specifications for these extensions can be found at: Vertex and Geometry Extensions GL_EXT_packed_float RGB floating-point textures in one 32bpp format GL_OES_mapbuffer Low-overhead buffer updates GL_OES_vertex_half_float 16-bit float vertex support (1 character bit, 5 exponent bits, 10 mantissa bits) FBO and Render Buffer extensions GL_OES_EGL_image Cross API images GL_OES_EGL_image_external Cross API images GL_OES_EGL_sync Command stream synchronization GL_OES_fbo_render_mipmap Mipmap-level FBO support GL_OES_rgb8_rgba8 24 and 32bpp-Fbos structural extensions GL_EXT_bgra Reversed RGBA Structure Support Support GL_EXT_texture_compression_dxt1 DXT1 Structure Support GL_EXT_texture_compression_latc LA Compressed Textures GL_EXT_texture_compression_s3tc DXT3/5 Texture Support GL_EXT_texture_format_BGRA8888 Reversed RGBA Texture Support GL_OES_compressed_ETC1_RGB8_texture ETC1 Textures GL_OES_texture_float Note that 32-bit floating point textures are accepted , but are converted internally into 16-bit floating point textures, and so the use of this extension is not recommended. GL_OES_texture_half_float 16-bit (1 character bit, 5 exponentbits, 10 mantissa bits) Texture Feature Extensions GL_EXT_texture_filter_anisotropic Anisotropic mipmap filtering GL_EXT_texture_array 1D arrays or 2D textures GL_EXT_unpack_subimage Limited stride support for texture updates GL_EXT_occlusion_query_boolean The list below highlights some of the NV-specific extensions and the links to their specifications in the Khronos registry. Some of these extensions are not yet included in the register; the specifications for these extensions are included at the end of the chapter. Other expansion Specs The following expansions specs are not yet in the Khronos registry site. They are included at the end of this chapter. GL_NV_draw_path GL_NV_shader_framebuffer_fetch NV_draw_path Name NV_draw_path Name Cords GL_NV_draw_path Contact Jussi Rasanen, NVIDIA Corporation (jrasanen 'at' nvidia.com) Tero Karras, NVIDIA Corporation (tkarras 'at' nvidia.com) Notice Copyright NVIDIA Corporation, ©2008 Status NVIDIA Proprietary Version Last Modified: 2008/09/16 NVIDIA Revision: 0.11 Number XXXX Not Yet XXXX Dependencies Written based on the article of the OpenGL 2.0 Specification. Requires OpenGL-ES 2.0.

Overview This extension adds functionality to flat display. Use cases for this extension include acceleration of vector image content and text rendering. A path is defined as a number of segments representing straight lines, or </KHR> </KHR> or cubic Bezier curves, and can be filled or caressed. Filling corresponds to generating the fragments that lie in the interior of the path. Stroking corresponds to generating the fragments that lie within an area defined by sweeping a straight pen along the path. Path segments are specified using a command array and a vertexcoördinate array. When a path is drawn, the command array is processed sequentily. There are two categories of commands: commands that draw a path segment, and categories that affect how the path appears. Depending on the type, each command consumes a variable number of coordinates of the vertexcoordinate matrix. When filling a path, the order in which the path segments are specified is disregarded. The only requirement is that they form zero or more closed contours. If a path contains undefined outlines, the interior and thus the resulting sequence of fragments is not defined. The contours of a path can have self-intersecting geometry and overlap with each other. For such paths, the interior is determined using a filling rule. Two fill lines, even-odd and non-zero, are provided. The direction of path segments is only important with the non-zero fill rule, as explained below. When petting a path, additional cap and join styles can be applied at the beginning and end of path segments. Joins are automatically generated between pairs of segments whose corresponding commands are next to each other. Caps are generated based on explicit path commands. Display quality can be controlled per path by specifying the maximum deviation from the ideal curve in window space. Padrendering Pipeline The padrender pipeline consists of three phases: transformation and texture coordinate generation, fill and line gridization, and fragment shading. This extension provides a minimal fixed function transformation and texture coordinates generation phase. Programmable vertexshaders are not supported in the context of pad rendering. Path Definition Paths are defined as a combination of an unchanging set of commands and an associated mutable sequence of vertex coordinates. Each command consumes zero or more vertex coordinates. Path commands appear as unsigned bytes, while the vertex coordinates data type for each path is specified separately. Toad vertices are always two-dimensional. The following table shows the available path commands: Path Command Coords Notes MOVE_TO_NV 2 Change the current position. LINE_TO_NV 2 Draw a straight line. QUADRATIC_BEZIER_TO_NV 4 Draw a quadratic Bezier curve. CUBIC_BEZIER_TO_NV 6 Draw a cubic Bezier curve. START_MARKER_NV 0 Take the current position. CLOSE_NV draw 0 Line to the recorded position. 0 Use cap style 0 in adjacent segment. STROKE_CAP1_NV 0 Use cap style 1 in adjacent segment. STROKE_CAP2_NV 0 Use cap style 2 in adjacent segment. STROKE_CAP3_NV 0 Use cap style 3 in adjacent segment. Transformation and structure coordinates generation path vertices specified by by vertexcoördinate order is converted by the transformation phase to the homogeneous shape (x, y, 0, 1) and then transformed from model space to clip space using the MATRIX_PATH_TO_CLIP_NV matrix. To facilitate texture mapping and color grading, the pad verticals are also transformed using each of the MATRIX_PATH_COORD[0-3] _NV matrices. A built-in fragmentcator varying array gl_PathCoord of type vec4 receives the corresponding interpolated values. The number of elements in the array gl_PathCoord is 4. While gradients and texture coordinates can also be implemented using the gl_FragCoord built-in fragmentcator variable, it is generally more efficient to use gl_PathCoord, avoiding unnecessary matrix multiplication per fragment. MATRIX_PATH_COORD[0-3]_NV and MATRIX_PATH_TO_CLIP_NV can define a homogeneous perspective transformation. It is up to fragment securing to normalize the interpolated coordinates if necessary. Fill a path When filling a path, the path segments must form zero or more closed outlines. If one of the contours remains open, the resulting set of fragments is not defined. This requirement can be reformulated as follows, depending on the value of FILL_RULE_NV: NON_ZERO_NV: Each two-dimensional point has an equal number of path segments that begin and end on it. EVEN_ODD_NV Each two-dimensional point has an even number of path segments that start or end on it. Please note that for two points to be considered identical, the binary representations of their coordinates must match exactly. To determine which slices to draw, the path commands are processed sequentially. The following temporary values are maintained during the process: i: Current vertex coordinate index, initially 0. cp: Current position, initially (0, 0). sp: Starting position, initially undefined. Each path command is processed, depending on the type as follows. c[i] is used to indicate the i'th value in the matrix of vertexcoördinate. MOVE_TO_NV: Replace the current position.cp = (c[i+0], c[i+1]), i += 2. LINE_TO_NV: Draw a straight line from <cp>to (c[i+0], c[i+1]).cp = (c[i+0], c[i+1]), i += 2. QUADRATIC_BEZIER_TO_NV: Draw a quadratic Bezier curve from <cp>to (c[i+2], c[i+3]) using(c[i+0], c[i+1]) as a checkpoint.cp = (c[i+2], c[i+3]), i+= 4. CUBIC_BEZIER_TO_NV: Draw a cubic Bezier curve from <cp>to (c[i+4], c[i+5]) using(c[i+0], c[i+1]) and (c[i+2], c[i+3]) as control points.cp = (c[i+4], c[i+5]), i+= 6. START_MARKER_NV: Replace the starting position.sp = cp. CLOSE_NV: If <sp>it is not defined, ignore the command. Otherwise, draw a straight line from <cp>to <sp>.cp = sp. STROKE_CAP[0-3]_NV: Ignore the assignment. START_MARKER_NV and CLOSE_NV assignments can be used to implement subpath closure found in many vector graphic content formats. For filled paths, an explicit LINE_TO_NV command to the starting position yields the same result as CLOSE_NV. For aligned paths, the difference is that CLOSE_NV the segment closing line to the segment</sp> </cp> </sp> </cp> </cp> </cp> Command. A fill rule is applied to determine whether a particular point is in the interior of the path. The fill lines are defined by projecting a radius from the point in question to infinity and counting the intersections of the radius and path segments. When you look along the direction of the beam, segments that cross from left to right take the counter, and the crosses from right to left take the counter. If the fill rule is NON_ZERO_NV, the point is within the interior if the final count is non-zero. If the fill line is EVEN_ODD_NV, the point is within the interior if the final count is odd. The counter must support at least 255 intersections. For more complex paths, the results are not defined. Curves can be accessed within a limit specified by the PATH_QUALITY_NV parameter. The limit defines the radius of a disc in the window space. By placing the disk at each sampling point, the following lines are used to determine whether the corresponding fragments should be generated: If the disk is fully in the path, you generate a snippet. If the disk is completely out of the path, you won't generate a snippet. If the disk is partially in the path, it's up to the deployment or generating a snippet to be generated. Stroking a path caressing is performed by sweeping a straight pen along each path segment, generating fragments for sampling points touched by the pen. In addition, cap and join styles can be applied at the beginning and end of the segments. Cap and join styles are selected for each path segment based on the path commands that border the commands that specify the segment and the values of the path parameters. The general rule is that the end of a slice is merged into the beginning of the next segment if specified by adjacent path commands. If the beginning or end of a segment is not merged, a limit is generated instead. Fill rule is not applied when stroking. Instead, a fragment is generated for each sampling point in the stroke. Even in the event that the stroke sweeps over a sampling point multiple times, only one fragment is generated. Dashing is not directly supported. Instead, this extension makes it possible to implement boisterous user code by generating the corresponding code. Paths are caressed in a coordinate space that differs from the path's user space and clip space. The transformation from the line space to the path's user space is controlled by the MATRIX_STROKE_TO_PATH_NV matrix. Both the path user space and the stroke space are two-dimensional, and so only the 2x2 upper left components of the matrix are used. consists of stroking a path out of five steps. First, the path segments are transformed from the path user space to the line space using the inverse of the line-to-path matrix. Second, the set of points affected by the stroke is determined in the line space, using a straight pen that extends one unit in each direction. Thirdly, the set of points is transformed from the line space back to the path user space using the Array. Fourth, the points are further transformed from the user space of the path to the clip space using the path-to-clip matrix. Fifth, a fragment is generated for each sampling point contained by the series of transformed points. The line-to-path matrix specifies the line width, regardless of how the path itself is transformed. Two common scenarios are line scales, where line width varies as the path-clip transformation changes, and non-scaling line, where the width remains constant in the clip space. To scale a line, the line-to-path matrix must be specified as an identity matrix multiplied by half the desired line width in the path user space. For non-scaling line, it must be specified as the inverse of the path clip matrix multiplied by half the line width in the clip space. The style of all joins is determined by the STROKE_JOIN_STYLE_NV path parameter, which can be set to one of the following values: JOIN_MITER_NV: Extend the incoming and outgoing line contours until they cross each other. If the distance between the intersection and the center is greater than STROKE_MITER_LIMIT_NV in the line space, apply an oblique join instead. JOIN_ROUND_NV: Connect the incoming and outbound line contours with a circular arc segment in the line space, corresponding to a single unit radius. JOIN_BEVEL_NV: connect the incoming and outgoing line contours with a straight line. JOIN_CLIPPED_MITER_NV: Same as JOIN_MITER_NV if STROKE_MITER_LIMIT_NV is not exceeded. Otherwise, cut the extended contours and connect them with a straight line. The cutting is done against a line whose distance from the centre is equal to STROKE_MITER_LIMIT_NV in the impact area and whose orientation is symmetrical with respect to the contours. The style of a start cap depends on the previous command path, and the style of a final cap depends on the next command. If the command is not STROKE_CAP[0-3]_NV, the cap style is STROKE_CAP_BUTT_NV. Otherwise, the style is determined by the corresponding STROKE_CAP[0-3]_STYLE_NV path parameter, each opf that can be set to one of the following values: CAP_BUTT_NV: End the segment with a straight line that connects the two perimeter endpoints. CAP_ROUND_NV: End the segment with a semicircle with a radius equal to one in the line space. CAP_SQUARE_NV: End the segment with a rectangle that extends one unit along the path line. CAP_TRIANGLE_NV: End the segment with a triangle with two vertices on the point outline ends and a third corner point one unit along the path line. As with fill, the path commands are processed sequentially, preserving the following temporary values: i: Current vertexcoördinate index, initially 0. cp: Current position, in authority (0, 0). ct: Current tangent, initially undefined. cs: Awaiting cap style, initially ass. sp: Starting position, initially undefined. st: Start tangent, initially undefined. Each command is processed, processed, processed as follows, on its type: MOVE_TO_NV: If <ct>is defined, draw a butt end cap on <cp>. cp = (c[i+0], c[i+1]), ct = undefined, cs = butt, i += 2. LINE_TO_NV, QUADRATIC_BEZIER_TO_NV, and CUBIC_BEZIER_TO_NV: Draw the segment that matches the command type. If <ct>is not defined, character a start cap of style <cs>on <cp>. If <ct>it is defined, draw a join between <cp>and the <ct>start line of the segment. If the previous command is START_MARKER_NV, replace <st>with the segment's starting line. cp = endpoint, ct = finalventrant, i += num. START_MARKER_NV: If <ct>it is defined, mark a butt end cap on <cp>. ct = undefined, cs = butt, sp = cp, st = undefined. CLOSE_NV: If <sp>it is not defined, ignore the command. Draw a straight line from <cp>to <sp>. If <ct>is not defined, character a start cap of style <cs>on <cp>. If <ct>it is defined, draw a join <cp>between and the direction of the <ct>line. If <st>is not defined, draw an ass end cap on <sp>. If <st>is defined, pull a join at sp between the direction of the line and <st>. cp = sp, ct = undefined, cs = butt. STROKE_CAP[0-3]_NV: If it is <ct>defined, draw a cap of the specified style. ct = undefined, cs = style specified by the command. Path programs are drawn using a special type of program object called a path program. Path programs work like normal program objects, except they don't allow a vertex shader to be specified. Path programs are created with a new CreatePathProgramNV() feature. Fragment Shader Fragment shader depth values are obtained by converting the homogeneous vertex coordinates (x, y, 0, 1) into the clip space. This allows the mixing of 3D and pad geometry using depth buffering. Since path programs do not support vertexshaders, path fragmentshaders cannot use user-defined different data. Instead, this expansion adds built-in variables gl_PathCoord[0-3] of type vec4 receiving interpolated vertex positions transformed with their respective MATRIX_PATH_COORD[0-3]_NV matrices. The value of gl_FrontFacing is not defined when rendering paths. The rest of the path When rendering paths, stencil functionality and backface culling are not applied. Mixing, dithering, depth testing, scissor testing, polygon shifting and multisampling are applied as with other primitives. Although stencil test and editing are not available when rendering paths, the original content of the stencil buffer is preserved. Pad buffers Pad buffers make an efficient display of animated text or other instanceed possible by allowing multiple path objects to be displayed with a single character call. A path buffer contains a list of path object handles and corresponding translation vectors. Invariance rules Changing path parameters, viewport, transformations, and clipping parameters can result in a different set of pixels appearing. New procedures and features uint CreatePath NV (enum datatype, sizei numCommands, const ubyte* commands);void DeletePath NV (uint path); void PathVerticesNV (uint</ct> </st> </st> </sp> </st> </ct> </cp> </ct> </cp> </cp> </c> </sp> </cp> </ct> </st> </ct> </cp> </ct> </cp> </cs> </ct> </cp> </ct>/ct> </ct> </ct> </ct>/ct> </ct> </ct> </ct> </ct>/ct> </ct> const void* vertices);void PathParameterfNV (uint path, enum paramType, float param); void PathParameteriNV (uint pad, enum paramType, int param); uint CreatePathProgram NV (invalid); PathMatrixNV(enum target, const float* value); void DrawPathNV (uint path, enum mode); uint CreatePath buffer NV (sizei capacity); annletionPath buffer NV (uint buffer); void PathbufferPathNV (uint buffer, int index, uint path); void PathbufferPositionNV (uint buffer, int index, float x, float y); void DrawPathbufferNV (uint buffer, enum mode); New types no new tokens accepted as parameter <paramType>from PathParameterNV: PATH_QUALITY_NV 0x8ED8 FILL_RULE_NV 0x8ED9STROKE_CAP0_STYLE_NV 0x8EE0 STROKE_CAP1_STYLE_NV 0x8EEE1 STROKE_CAP2_STYLE_NV 0x8EEE2 STROKE_CAP3_STYLE_NV 0x8EE3 STROKE_JOIN_STYLE_NV 0x8EE8 STROKE_MITER_LIMIT_NV 0x8EEE9 Pad para ILL_RULE_NV meter values: EVEN_ODD_NV 0x8EF0NON_ZERO_NV 0x8EF1 values for the CAP parameter[0-3]_STYLE_NV : CAP_BUTT_NV 0x8EF4CAP_ROUND_NV 0x8EF5 CAP_SQUARE_NV 0x8EF6 CAP_TRIANGLE_NV 0x8EF7 pad para JOIN_STYLE_NV meter values: JOIN_MITER_NV 0x8EFCJOIN_ROUND_NV 0x8EFDJOIN_BEVEL_NV 0x8EFEJOIN_CLIPPED_MITER_NV 0x8EFF Accepted as <target>PathMatrixNV parameter: MATRIX_PATH_TO_CLIP_NV 0x8F04MATRIX_STROKE_TO_PATH_NV 0x8F05MATRIX_PATH_COORD0_NV 0x8F08MATRIX_PATH_COORD1_NV 0x8F09MATRIX_PATH_COORD2_NV 0x8F0AMATRIX_PATH_COORD3_NV 0X8F0B Accepted as <mode>DrawPath bufferNV parameter : FILL_PATH_NV 0x8F18STROKE_PATH_NV 0x8F19 Accepted as path commands by CreatePathNV: MOVE_TO_NV 0x00LINE_TO_NV 0x01QUADRATIC_BEZIER_TO_NV 0x02CUBIC_BEZIER_TO_NV 0x03START_MARKER_NV 0x20CLOSE_NV 0x21STROKE_CAP0_NV 0x40STROKE_CAP1_NV 0x41STROKE_CAP2_NV 0x42STROKE_CAP3_NV 0x43 Additions to Chapter 2 of the OpenGL ES specification Add the following error conditions to Chapter 2.8 under DrawArrays : An error INVALID_OPERATION generates if the current program is a path program. Add the following error conditions to Chapter 2.8 under DrawElements: A INVALID_OPERATION error is generated if the current program is a path program. Add the following error terms to Chapter 2.15 under AttachShader. A INVALID_OPERATION error is generated if the program is a path program and the shader is a corner point shader. Add the following error conditions to Chapter 2.15 under LinkProgram. Linking a program without a vertex shader will not fail if the program is a path program. Additions to Chapter 3 of the OpenGL ES Specification Add a new section between sections 3.5 (Polygons) and 3.6 (Pixel Rectangles) 3.6 Paths This extension adds a new type of primitive paths to the primitives of OpenGL ES - points, lines, polygons, pixel angles, and bitmaps. 3.6.1 Path objects The call creates new path objects: uint CreatePath NV (enumerated datatype, sizei const ubyte* commands); when <datatype>is the data type of the corner point and it is one of [UNSIGNED_]BYTE, [UNSIGNED_]SHORT, [UNSIGNED_]INT, FLOAT,</datatype> </mode> </target> </paramType> </paramType> <numCommands>is the number of commands in the path definition and <commands>is a pointer to an unsigned command bytearray. Valid assignments are listed below. The function returns a non-zero handle to the object or 0 in case of error. A INVALID_ENUM error is generated if <datatype>this is not one of the values specified above. A INVALID_VALUE error is generated if <numCommands>it is less than zero or greater than zero and is NULL or the array <numCommands>contains a <commands> <commands>invalid command. TODO notes that path objects can be shared between multiple contexts. Path objects are removed with the void DeletePathNV (uint path) command; where <path>is the handle of the path object that needs to be removed. If the path is assigned to one or more path buffers, path resources are released only when the last reference to the path is removed. Path handle is invalid after a call to DeletePathNV. A INVALID_VALUE error is generated if the path object does not exist. Path vertices are specified with the command void PathVerticesNV (uint pad, const void * vertices); where <path>is the handle to the path object and <vertices>is a pointer to an array of vertices. <vertices>must contain at least as many coordinates as consumed by the corresponding path commands, otherwise the results are not defined and may lead to a program crash. If <vertices>the rest contains more coordinates than the path commands, the rest is quietly ignored. A INVALID_VALUE error is generated if the specified <path>object does not exist or if it is NULL and requires the <vertices>vertices path command vertices. Path parameters are set using the commands void PathParameterfNV (uint pad, enum paramType, float param); void PathParameteriNV (uint pad, enum paramType, int param); where <path>is the path object handle, the parameter is set and is the value of the <paramType> <param> parameter. The following symbols are accepted as <paramType>: PATH_QUALITY_NV Maximum allowed deviation from the ideal path measured in pixels. The default value is 0.5 pixels. FILL_RULE_NV use Fill rule to fill paths. <param>must be EVEN_ODD_NV or NON_ZERO_NV. The default value is EVEN_ODD_NV. STROKE_CAPn_STYLE_NV Cap style for the cap index n used when stroking a toad. The defaults are CAP_BUTT_NV. STROKE_JOIN_STYLE_NV Join style used when petting a path. The default value is JOIN_MITER_NV. STROKE_MITER_LIMIT_NV Miter limit used when petting a path with miter joins. If a join angle exceeds the limit, a miter join is converted to an oblique side join. The default value is 4. As a paramType, PATH_QUALITY_NV is in is converted into a float. If paramType is not PATH_QUALITY_NV in PathParameterfNV(), param is converted to an int. A INVALID_VALUE error is generated if the <path>object does not exist. A INVALID_ENUM error is generated as <paramType>is not one of the above. A INVALID_VALUE error is generated as <paramType>PATH_QUALITY_NV <param> and <= 0= or=></=> <paramType></paramType> </paramType> </paramType> </path> </paramType> </paramType> </path> </vertices> </vertices> </path> </path> </commands> </commands> </numCommands&gt ; </numCommands> </datatype> </commands></numCommands> </commands></numCommands> STROKE_MITER_LIMIT_NV and <param> < 1.= an= invalid_enum= error= is= generated= if=> <paramType>is FILL_RULE_NV and param is not a valid fill rule, <paramType>is STROKE_CAPn_STYLE_NV and is not a valid cap <param> style, or is STROKE_JOIN_STYLE_NV and <paramType>is not a valid join <param> style. Path transformations are set using the value of the call void PathMatrixNV(enum target, const float* value); where <value>a 4x4 matrix must be specified. The following values are accepted as <target>parameter: MATRIX_PATH_TO_CLIP_NV used to convert path verticals into clipspace when drawing a path. MATRIX_STROKE_TO_PATH_NV used to transform the pen when stroking a toad. The vertices are then converted into clamp space by MATRIX_PATH_TO_CLIP_NV matrix. Only the 2x2 submatrix in the upper left is used. MATRIX_PATH_COORDn_NV Used to generate values for gl_PathCoord[0-3] varies for fragment shader by transforming corner point positions. The default value for all matrices is the identity matrix. A INVALID_ENUM error is generated as <target>is not one of the above. A INVALID_VALUE error is generated if the value is NULL. A path is displayed using the call void DrawPathNV (uint path, enum mode); where <path>is to draw the path and should be FILL_PATH_NV or <mode>STROKE_PATH_NV. A INVALID_VALUE error is generated if it does not exist <path>. A INVALID_OPERATION error is generated if there is no current program, the current program is not a path program, the stencil test is enabled, the polygon mode is not GL_FILL or the shadow model is not GL_SMOOTH. A INVALID_ENUM is generated as <mode>there is no FILL_PATH_NV or STROKE_PATH_NV. 3.6.2 Path Programs A path program is a special type of program object that behaves differently to a normal program object, but can only confirm fragment searceration. Path programs are created with the Uint CreatePathProgramNV command(void) The function returns 0 on error (i.e. OUT_OF_MEMORY). TODO describe LinkProgram error conditions here for clarity? 3.6.3 Pad buffers Pad buffers can be used to efficiently display multiple instances of a set of path objects with a single character call. Each path in a path buffer has an associated positive ctor that can specify a shift of the model space position for that path. Path buffers are created using the uint CreatePath bufferNV(sizei capacity) function where <capacity>the number of paths in a path buffer is. This feature returns a non-zero handle to a path buffer object or 0 in case of error. A INVALID_VALUE error is generated if the capacity < 0.= todo= note= that= path= buffer= objects= can= can= be= shared= multiple= path= buffers= are= deleted= using= the= call= void= deletepath buffernv(= uint= buffer=);= where=> <buffer>the handle is for the pad buffer. Path buffer hold is invalid after a call to DeletePath bufferNV. A INVALID_VALUE error is generated if the path buffer does not exist. A path can be added to or removed from a pad buffer with the PathbufferPathNV void function, uint buffer, int index,</buffer> </capacity> </mode> </path> </mode> </path> </target> </target> </value> </paramType> </paramType> path); where <buffer> is the path buffer object handle, the index of the path buffer <index> slot and <path> is the path object handle. Path buffer paths are mutabel and can be respecified later. Call Path bufferPathNV with <path> set to zero removes path from the path buffer and leaves the slot corresponding to <index> empty. A INVALID_VALUE error is generated if the path buffer object <buffer> does not exist or is less than zero or greater than or equal to the <index> path buffer capacity. Pad buffer pad positive vector is specified using the path buffer launch Path buffer, int index, float x, float y ; where <buffer> is the pad buffer object handle, <index> is the pad buffer slot index and <x> and enter the translation <y> Pad buffer path translations are mutabel and can be respecified later. A INVALID_VALUE error is generated if the path buffer does not exist or indexes the capacity < 0= or= index=> of the path buffer. All paths in a toad buffer are displayed using the command void DrawPath bufferNV (uint buffer, enum mode); A INVALID_VALUE error is generated if <buffer> it does not exist. A INVALID_OPERATION error is generated if there is no current program, the current program is not a path program, the stencil test is enabled, the polygon mode is not GL_FILL or the shadow model is not GL_SMOOTH. A INVALID_ENUM is generated as <mode> there is no FILL_PATH_NV or STROKE_PATH_NV. The effect of a DrawPath bufferNV call is the same as when DrawPathNV was called for each individual path reference in the path buffer, arranged from the first index to the last. </mode></buffer></y></x><<<</buffer></index></buffer><<<<</buffer><>/buffer><</buffer>&/buffer>>>>/buffer>>>>>>/buffer>>>>>