3D Pointer

This extension provides the ability to define 2D pointers in the window space as well as 3D projections in the world space. The extension can then convert the 2D pointer into 3D world space, as well as check for collisions through 3D polygons. This makes 3D object interaction extremely simple. Because this system supports multiple pointers, projections, and viewport properties you can have as many 2D positions converted to 3D space over any number of 3D camera views at various screen locations.

Scripts

m3d_pointer_create()

Description: *Creates a new 2D pointer structure to be used in the system alongside a projection.*
Returns: *(real) id of the pointer, needed in other functions*

m3d_pointer_setpos(id, x, y)

Description: *Sets the position of the specified pointer in 2D window coordinates.*
Argument0: *(real) id of the pointer to modify*
Argument1: *(real) x-position value*
Argument2: *(real) y-position value*
Returns: *(undefined) N/A*

m3d_pointer_get_x(id)

Description: *Returns the current x-position in window coordinates for the specified pointer.*
Argument0: *(real) id of the pointer to use*
Returns: *(real) x-position in window coordinates*

m3d_pointer_get_y(id)

Description: *Returns the current y-position in window coordinates for the specified pointer.*
Argument0: *(real) id of the pointer to use*
Returns: *(real) y-position in window coordinates*

m3d_pointer_get_z(id)

Description: *Returns the current z-position in window coordinates for the specified pointer.*
Argument0: *(real) id of the pointer to use*
Returns: *(real) z-position in window coordinates*

m3d_pointer_destroy(id)

Description: *Destroys the specified pointer and all associated data.*

Argument0: *(real) id of the pointer to destroy*
Returns: *(undefined) N/A*

m3d_projection_create(xport, yport, wport, hport)
Description: *Creates a new projection structure for later use with the system.*
Argument0: *(real) x-position of the projection in window coordinates*
Argument1: *(real) y-position of the projection in window coordinates*
Argument2: *(real) width of the projection in window dimensions*
Argument3: *(real) height of the projection in window dimensions*
Returns: *(real) id of the projection, needed in other functions*

m3d_projection_define(id, xfrom, yfrom, zfrom, xto, yto, zto, xup, yup, zup, angle, aspect)
Description: *Defines the 3D properties of the projection. If you decide to manually draw your 3D projection separately, the values of that projection should be mimicked when calling this function.*
Argument0: *(real) id of the projection to use*
Argument1: *(real) x-position of the camera in world coordinates*
Argument2: *(real) y-position of the camera in world coordinates*
Argument3: *(real) z-position of the camera in world coordinates*
Argument4: *(real) x-position of the target in world coordinates*
Argument5: *(real) y-position of the target in world coordinates*
Argument6: *(real) z-position of the target in world coordinates*
Argument7: *(real) x-value of the up vector*
Argument8: *(real) y-value of the up vector*
Argument9: *(real) z-value of the up vector*
Argument10: *(real) Angle, or horizontal FOV for the projection*
Argument11: *(real) Aspect ratio of the projection*
Returns: *(undefined) N/A*

m3d_projection_calculate(projection id, pointer id)
Description: *Converts a pointer into a 3D directional vector, based on the projection specified. This only needs to be called once per pointer per step where a location is needed.*
Argument0: *(real) id of the projection to use*
Argument1: *(real) id of the pointer to use*
Returns: *([1D ARRAY] real) formatted array containing 4 values in the following positions:*
0 - x-value of directional vector
1 - y-value of directional vector
2 - z-value of directional vector
3 - id of projection

m3d_projection_draw(id, near, far)

Description: *This is an optional function that will draw the m3d projection as a d3d projection on the screen to remove the necessity of defining the properties twice, once for m3d, and one for d3d_projection_*.*

Argument0: *(real) id of the projection to use*

Argument1: *(real) near clipping plane in world units*

Argument2: *(real) far clipping plane in world units*

Returns: *(undefined) N/A*

m3d_projection_destroy(id)

Description: *Destroys the specified projection and all associated data.*

Argument0: *(real) id of the projection to destroy*

Returns: *(undefined) N/A*

m3d_projection_collision_get_x(id)

Description: *If there was a collision between a pointer and 3D object (see below functions), the coordinates of the collision will be stored in the projection.*
This function returns the x-position in world coordinates.
If no collision has ever occurred, 0 is returned.
If the previous check had no collision, the last collision coordinate will be returned.

Argument0: *id of the projection to check*

Returns: *(real) x-position in world coordinates*

m3d_projection_collision_get_y(id)

Description: *If there was a collision between a pointer and 3D object (see below functions), the coordinates of the collision will be stored in the projection.*
This function returns the y-position in world coordinates.
If no collision has ever occurred, 0 is returned.
If the previous check had no collision, the last collision coordinate will be returned.

Argument0: *id of the projection to check*

Returns: *(real) y-position in world coordinates*

m3d_projection_collision_get_z(id)

Description: *If there was a collision between a pointer and 3D object (see below functions), the coordinates of the collision will be stored in the projection.*
This function returns the z-position in world coordinates.
If no collision has ever occurred, 0 is returned.
If the previous check had no collision, the last collision coordinate will be returned.

Argument0: *id of the projection to check*

Returns: *(real) z-position in world coordinates*

m3d_intersect_triangle([1D ARRAY] pVec, x1, y1, z1, x2, y2, z2, x3, y3, z3)

Description: *Checks if a pointer has a collision with a 3D triangle. If true, the coordinates of the collision will be recorded in the projection and can be retrieved with the functions listed above.*

Argument0: *([1D ARRAY] real) a 1D vector containing a directional vector and projection id. This vector is returned by the function* m3d_projection_calculate.

Argument1: *(real) x-coordinate of the first point in world coordinates*

Argument2: *(real) y-coordinate of the first point in world coordinates*

Argument3: *(real) z-coordinate of the first point in world coordinates*

Argument4: *(real) x-coordinate of the second point in world coordinates*

Argument5: *(real) y-coordinate of the second point in world coordinates*

Argument6: *(real) z-coordinate of the second point in world coordinates*

Argument7: *(real) x-coordinate of the third point in world coordinates*

Argument8: *(real) y-coordinate of the third point in world coordinates*

Argument9: *(real) z-coordinate of the third point in world coordinates*

Returns: *(bool) if true, a collision occurred*

m3d_intersect_plane([1D ARRAY] pVec, vx1, vy1, vz1, vx2, vy2, vz2, px, py, pz)

Description: *Checks if a pointer has a collision with an infinite 3D plane. If true, the coordinates of the collision will be recorded in the projection and can be retrieved with the functions listed above.*
This function requires two different non-parallel vectors that lie parallel to the plane.

Argument0: *([1D ARRAY] real) a 1D vector containing a directional vector and projection id. This vector is returned by the function m3d_projection_calculate.*

Argument1: *(real) x-value of first vector*

Argument2: *(real) y-value of first vector*

Argument3: *(real) z-value of first vector*

Argument4: *(real) x-value of second vector*

Argument5: *(real) y-value of second vector*

Argument6: *(real) z-value of second vector*

Argument7: *(real) x-position of point resting on the plane*

Argument8: *(real) y-position of point resting on the plane*

Argument9: *(real) z-position of point resting on the plane*

Returns: *(bool) if true, a collision occurred*

Getting Started

The most basic application of this extension is to have clickable objects in 3D space. In order to do this, we need to know the projection data, aka where the player is looking, as well as the 2D pointer data, aka where the user's mouse is located on the screen.

These two requirements have been converted into structures that can then be passed into scripts to perform the difficult math for you.

In the above situation, we would need to create one pointer object for the mouse, and one projection object for the 3D camera. Once this is done, we can check against any triangle or infinite plane at any time.

An example initialization would be as follows:

```
_pointer = m3d_pointer_create();
_projection = m3d_projection_create(0, 0, window_get_width(), window_get_height());
```

This creates the pointer object, as well as the projection that will be used for the conversion. We specify that the projection takes up the entire window. In most cases, this is what you want. Because we want the pointer to follow the mouse, we will need to update it every step with the new mouse position:

```
m3d_pointer_setpos(_pointer, window_mouse_get_x(), window_mouse_get_y());
```

Let's assume we have a static camera for this example and want to grab the mouse's position on the z=0 plane. In the draw event, we could set it up something like this:

```
// We set up the projection just like we would a normal 3D camera:
m3d_projection_define(_projection, -128, 0, 24, 0, 0, 0, 0, 0, 1, 60, 16/9);

// This optionally renders the scene. You can also use d3d_projection* instead.
m3d_projection_draw(_projection, 1, 1024)

// Next, we must convert the mouse to a directional 3D vector based
// on the projection:
var __vec = m3d_projection_calculate(_projection, _pointer);

// Once we have the vector, we can pass it into the desired plane and see
// if it collides:
var __collision = m3d_intersect_plane(__vec, 1, 0, 0, 0, 1, 0, 0, 0, 0);
if (__collision) {  // If a collision, let's draw an ellipsoid at the mouse's position:
        var __x, __y, __z;
        __x = m3d_projection_collision_get_x(_projection);
        __y = m3d_projection_collision_get_y(_projection);
        __z = m3d_projection_collision_get_z(_projection);
        d3d_draw_ellipsoid(__x - 2, __y - 2, __z - 2, __x +2, __y +2, __z + 2, -1, 1, 1, 8);
}
```