Simple 3D Library

Introduction

The Simple 3D library (s3d) lets you set up a 3D view and render triangles and quadrilaterals. It is a software renderer, meaning that it doesn't use DirectX or OpenGL – everything is computed on the CPU (this goes for all drawing in n7). So don't expect high performance, at least not in this first test version. There are countless optimizations to be made.

Transformations

If you have used turtle graphics you shouldn't have much problems understanding how to position objects in 3D. And if you've played around with legacy OpenGL you probably don't need to read this at all. A position in 3D space is defined by an x, y and z coordinate, (x, y, z). Imagine that you're standing at the position (0, 0, 0). The x axis points right, the y axis down and the z axis forward (atleast in s3d). To position something, let's say a box, at position (0, 0, 4) (four ”units” in front of you) and rotate it 45 degrees around its y-axis, it's easiest to imagine it as if you walk to that position first, spin around 45 degrees and then put the box down. The walk is called a translation, and the spinning is called a rotation. In s3d you would use the function call S3D_Translate(0, 0, 4) followed by S3D_RotateY(rad(45)) to get to the desired position of the box. And if the box was a mesh (3D model) that you created or loaded, you would then put it down with S3D_Mesh(myBox, 0) (don't mind the second parameter for now). There's a third transformation called scaling, with which you can change the size of an object. If you had wanted to make the box twice as tall you could have called S3D_Scale(1, 2, 1) before putting it down with S3D_Mesh. Walking can be exhausting. But instead of rotating 45 degrees in the opposite direction and walk back to position (0, 0, 0) after putting down the box, you can use the transformation stack to teleport back. If you call S3D_Push() before S3D_Translate(0, 0, 4), your position and direction will be saved. And when you've put the box down with S3D_Mesh(myBox, 0) you can call S3D_Pop() to teleport back to where you were. After teleporting back you can keep walking, spinning and teleporting to place other objects in space.

Functions

S3D_SetView((number)*tagetImage*, (number)*fov*, (number)*zMin*, (number)*zMax*)

Set up the rendering view. *targetImage* is the image you want to render to, in most cases you set it to *primary* (the window). *fov* is the vertical field of view in radians, and *zMin* and *zMax* are the near and far clip planes. Anything rendered at a distance smaller than *zMin* or greater than *zMax* will be clipped or not rendered at all.

S3D_SetPerspectiveCorrection((number)*mode*)

Set the perspective correction quality for texture mapping to *mode*, which should be one of these constants.

S3D_NONE

Don't use this

S3D_FAST (default)

Bad perspective correction

S3D_NORMAL

Less bad perspective correction

S3D_NICE

Okay perspective correction

S3D_SetDepthBuffer((number)*mode*)

The only type of depth buffer that s3d currently supports is a z buffer. When a z buffer is used, a depth value is stored for each pixel of the target image. Every time a new pixel is to be rendered, its distance is compared with the distance stored in the z buffer. The new pixel is only drawn if its distance (to the camera) is less than what is already stored in the buffer. If it is rendered, the value in the depth buffer is changed to that of the new pixel. *mode* should be one of these constants:

S3D_NONE

Don't use the z buffer at all

S3D_Z_BUFFER (default)

Compare depths and only draw and update the z buffer if the new pixel is closer to the camera

S3D_Z_WRITE

Don't compare depths, draw and update the z buffer no matter what

S3D_Z_READ

Compare depths and only draw if the new pixel is closer, but don't update the z buffer

S3D_ClearDepthBuffer()

Clear the depth buffer, meaning that the depth of each pixel is set to the highest possible distance.

S3D_SetSorting((number)*mode*)

Before rendering a batch of polygons, you can automatically sort them based on their average distance to the camera. In that case, nothing is actually rendereded until you call *S3D_Render*. Some times, depending on the layout of the polygons, you can use this instead of a depth buffer (N7/examples/libraries/s3d/ex_heightmap.n7 is a good example of this). *mode* should be one of these constants:

S3D_NONE (default)

No sorting, all polygons are rendered directly

S3D_BACK_TO_FRONT

Render from highest to lowest distance

S3D_FRONT_TO_BACK

Render from lowest to highest distance

S3D_ClearTransformation()

Clear the current transformation (teleport to position (0, 0, 0) and look along the positive z axis, with the x axis pointing stright to the right and the y axis pointing down).

S3D_Clear()

Clear the current transformation and reset the transformation stack, clear the depth buffer and set the (s3d) draw color to opaque white (255, 255, 255, 255).

S3D_Translate((number)*x*, (number)*y*, (number)*z*)

Translate by (*x*, *y*, *z*) relative to the current transformation.

S3D_RotateX((number)*a*)

Rotate *a* radians around the x axis relative to the current transformation.

S3D_RotateY((number)*a*)

Rotate *a* radians around the y axis relative to the current transformation.

S3D_RotateZ((number)*a*)

Rotate *a* radians around the z axis relative to the current transformation.

S3D_Scale((number)*x*, (number)*y*, (number)*z*)

Scale x, y and z coordinates by (*x*, *y*, *z*) relative to the current transformation.

S3D_Push()

Push the current transformation to the stack.

S3D_Pop()

Pop transformation from the stack and make it the current.

S3D_Begin((number)*mode*)

Start defining polygons (or faces) of the type *mode* to be rendered. *mode* should be one of the following constants:

S3D_TRIANGLES

Define triangles, each triangle requires three calls to *S3D_Vertex*

S3D_QUADS

Define quadrilaterals, each quad requires four calls to *S3D_Vertex*

If you haven't called *S3D_SetSorting* to sort your polygons, a triangle or quad will be rendered directly after each third or fourth call to *S3D_Vertex*. If you have enabled sorting, the polygons won't be rendered until you call *S3D_Render*.

S3D_End()

Stop rendering polygons of the type specified when calling *S3D_Begin*. Each *S3D_Begin* must be matched by an *S3D_End* and nesting is not allowed.

S3D_Vertex((number)*x*, (number)*y*, (number)*z*, (number)*u*, (number)*v*)

S3D_Vertex4((number)*x*, (number)*y*, (number)*z*, (number)*u*, (number)*v*)

Specify a vertex (position in space), (*x*, *y*, *z*), and texture coordinates, (*u*, *v*), for a for a triangle or a quad. Texture coordinates are always in the range [0..1]. (0, 0) represents the top left corner of an image, and (1, 1) is the bottom right corner. This makes it easy to use textures of different sizes without having to change the texture coordinates. Note that the order in which you add the vertices matter! If a polygon faces away from the camera, it won't be rendered. Let's say you want to render a square of the size 1x1 centered in the xy plane at a distance of 3 units and facing the camera. Then you need to add the vertices in clockwise order (remember that the x axis points right and the y axis down):

```
S3D_Vertex(-0.5, -0.5, 3, 0, 0)

S3D_Vertex(0.5, -0.5, 3, 1, 0)

S3D_Vertex(0.5, 0.5, 3, 1, 1)

S3D_Vertex(-0.5, 0.5, 3, 0, 1)
```

Learning to ”see” this can take some time, but hopefully you will get used to it.

S3D_Vertex3((number)*x*, (number)*y*, (number)*z*)

Same as *S3D_Vertex*/*S3D_Vertex4* but texture coordinates are set to 0.

S3D_Color((number)*r*, (number)*g*, (number)*b*)

S3D_Color3((number)*r*, (number)*g*, (number)*b*)

Set the color for all coming polygons to *r*, *g*, *b*, where all parameters should be in the range [0..255]. If the polygons are textured, they will be colorized/tinted (just like regular images when using *set color* and *draw image*). Note that drawing textured polygons that are colorized is quite a bit slower than drawing non-colorized textured polygons. Use *S3D_Color*(255, 255, 255) to disable colorization.

S3D_Color4((number)*r*, (number)*g*, (number)*b*, (number)*a*)

Same as *S3D_Color*/*S3D_Color3* but also set's an alpha (opacity) component in the range [0..255]. When using S3D_Color/S3D_Color3, the alpha component is 255. Transparency should be used with great care - none-opaque polygons must be rendered back to front.

S3D_Additive((number)*value*)

Enable additive color blending if *value* is *true*, disable if *false*.

S3D_Texture((number)*img*)

Set the texture to use for all coming polygons to image *img*, a normal n7 image id.

S3D_Render()

Sort and render polygons.

S3D_RenderFog((number)*r*, (number)*g*, (number)*b*, (number)*retro*)

If you have rendered your polygons using a depth buffer, in the mode *S3D_Z_BUFFER* or *S3D_Z_BUFFER_WRITE*, you can apply a fog effect to the scene with this function. It uses the depth values in the buffer to modify the color of all rendered pixels. If a pixel has the same depth as the *zMax* value passed to *S3D_SetView*, it will be fully replaced by the color (*r*, *g*, *b*), while pixels closer to the camera will be less affected. If *retro* is set to *true*, the fog will have a retro look (sharp, distinct, levels).

(number)**S3D_LoadMesh**((string)*filename*, (number)*scaleX*, (number)*scaleY*, (number)*scaleZ*, (number)*invertFaces*)

Load a mesh from the file *filename*. The returned identifier can be used when calling *S3D_Mesh*, *S3D_BlendMesh* and *S3D_FreeMesh*. Currently s3d only supports loading meshes (models) from OBJ files (but you can find an md2, Quake 2 model, loader on the naalaa forum). When it comes to materials, it uses the diffuse colors (Kd) and textures (map_Kd) from the material file (MTL). Since the coordinate system of s3d doesn't match (by default) that of OpenGL (I went the JavaFX way), you may need to apply scaling to the models you load using *scaleX*, *scaleY* and *scaleZ*. Usually *scaleY* should be negative. If the faces looks all weird (you see the inside of a mesh rather than its outside) you should set *invertFaces* to *true* (I haven't had to do that for any model yet). You can also use *scaleX*, *scaleY* and *scaleZ* to resize a model once, rather than calling *S3D_Scale* everytime you render it. The function returns an *unset* variable if the mesh could not be loaded, but most likely it will cause a runtime error if there's something weird with the file. I must remind you that s3d uses rather unoptimized software rendering. So the models you load and use should have really, REALLY, low polygon counts. Look at the first Quake game (not the remastered version) to understand what I mean! When people today talk about ”low poly” models, they're like: ”Yeah, this tiny rock I made only has 5,983 polygons! It's so retro!” That rock alone would break s3d.

(number)**S3D_LoadMeshFrame**((number)*mesh*, (string)*filename*, (number)*scaleX*, (number)*scaleY*, (number)*scaleZ*)

S3d supports blending of mesh frames for smooth animation, but OBJ files has no support for frames or animation. As a workaround you can use *S3D_LoadMeshFrame* to load the vertices of separate OBJ files and add them as animation key frames to a previously loaded mesh. This, of course, requires that the OBJ files represent the same object and have exactly the same number of vertices in the same order. *mesh* should be an already loaded mesh, and the parameters should correspond to those used when loading that mesh. The function returns *true* on success.

S3D_Mesh((number)*mesh*, (number)*frame*)

Render the mesh *mesh* with the current transformation. If the mesh has more than one frame, you select which frame to render with the *frame* parameter. It should be set to 0 for the original mesh, 1 for the second frame added and so on.

S3D_BlendMesh((number)*mesh*, (number)*frame0*, (number)*frame1*, (number)*blend*)

If a mesh has several frames, you can render it by blending the vertices of two frames. The original mesh has the frame index 0, the second frame added has index 1 and so on. *blend* should be in the range [0..1]. If *blend* is 0, the mesh will be rendered as frame *frame0*, and if it's 1 it will be rendered as *frame1*. If *blend* is 0.75, the mesh will look 25% like *frame0* and 75% like *frame1*.

(number)**S3D_BeginMesh**()

Rendering a mesh with *S3D_Mesh* is a lot faster than using *S3D_Begin*/*S3D_End* and defining all the vertices one by one every frame of your game loop. *S3D_Mesh* performs optimizations and can discard polygons at an early stage. You can use *S3D_BeginMesh* and *S3D_EndMesh* to ”record” a mesh that you can then render with *S3D_Mesh*/*S3D_BlendMesh*. *S3D_BeginMesh* returns the mesh identifier. Inbetween *S3D_BeginMesh* and *S3D_EndMesh* you can use *S3D_Begin*/*S3D_End*, *S3D_Vertex*, *S3D_Color* and all the transformation functions. Look at the square from the explanation of the *S3D_Vertex* function:

```
S3D_Vertex(-0.5, -0.5, 3, 0, 0)

S3D_Vertex(0.5, -0.5, 3, 1, 0)

S3D_Vertex(0.5, 0.5, 3, 1, 1)

S3D_Vertex(-0.5, 0.5, 3, 0, 1)
```

You can record that mesh and render as many instances of it as you like in your game loop. But let's set the z coordinate to 0, so that we can use *S3D_Translate* to render it at different locations:

```
squareMesh = S3D_BeginMesh()
    S3D_Begin(S3D_QUADS)
        S3D_Vertex(-0.5, -0.5, 0, 0, 0)
        S3D_Vertex(0.5, -0.5, 0, 1, 0)
        S3D_Vertex(0.5, 0.5, 0, 1, 1)
        S3D_Vertex(-0.5, 0.5, 0, 0, 1)
    S3D_End()
S3D_EndMesh()
```

You can even draw other meshes while recording a mesh.

S3D_EndMesh()

Stop recording of mesh started with S3D_Begin.

(number)**S3D_CreateMesh**((array)*vertexList*, (array)*uvList*, (array)*materialList*, (array)*faceList*)

This function is used internally by *S3D_LoadMesh* and shouldn't really be used by sane people. It lets you create a mesh by supplying arrays with data for vertices, texture coordinates, materials and faces (polygons). It returns a mesh on success. The parameters are explained here:

vertexList

Array of vertex positions: [[x0, y0, z0], [x1, y1, z1], ..] Must not be *unset* or empty

uvList

Array of texture coordinate pairs: [[u0, v0], [u1, v1], ..] Must not be *unset* or empty. Set to [[0, 0]] if there are no texture coordinates! Yes, this is sort of a bug.

materialList

Array of materials: [[[r0, g0, b0[, a0]], texture0], [[r1, g1, b1[, a1]], texture1], ..] May be *unset* or empty. An rgb/rgba array may be set to *unset*, and the same goes for a texture

faceList

A list of faces: [[vertexIndex00, vertexIndex01, vertexindex02[, vertexIndex03], uvIndex00, uvIndex01, uvIndex02[, uvIndex03], materialIndex0], [vertexIndex10, vertexIndex11, vertexindex12[, vertexIndex13], uvIndex10, uvIndex11, uvIndex12[, uvIndex13], materialIndex1], ..] vertexIndex refers to an index in *vertexList*, uvIndex to *uvList* and materialIndex to *materialList*. Triangles require 3 vertex and uv indexes and a quad 4.

materialindex should be set to unset if no material is used. uvIndex may not be unset

S3D_AddMeshFrame((number)*mesh*, (array)*vertexList*)

This function is used internally by *S3D_LoadMeshFrame*. It adds a frame to the mesh *mesh*. *vertexList* must contain the same number of vertices as in the original mesh, and it must have the same format as the *vertexList* parameter of *S3D_CreateMesh*.

S3D_FreeMesh((number)*mesh*)

Dispose the mesh *mesh*.

S3D_TransformVector((array)*dst*, (array)*src*)

Multiplies the vector *src*, [x, y, z[, w = 1]], with the current transformation matrix and writes the resulting vector to *dst*.

(number)**S3D_ProjectVector**((array)*dst*, (array)*src*)

Tries to transform and project the vector *src*, [x, y, z[, w = 1]], using the current transformation and projection matrices and writes the resulting vector to *dst* on success. The function will fail and return *false* if the transformed z coordinate is less than the near clipping plane. On success the x and y coordinates of dst will be in screen coordinates (they may, however, be < 0 or >= the width of the target image). I guess this function can be used for visibility tests.

(number)**S3D_ProjectFace**((array)*dst*, (array)*src*)

Tries to project the polygon *src* using the current transformation and projection matrices and writes the resulting polygon to *dst* on success. The function returns the number of vertices in the resulting polygon. If it returns 0, the transformed z coordinates of all vertices were less than the near clipping plane. On success, ther number of vertices in *dst* may be more or less than those in *src* due to clipping against the near clipping plane. The format av *src* should be: [x0, y0, z0, x1, y1, z1, …], and the number of points should be either 3 or 4. On success the x and y coordinates of *dst* will be in screen coordinates (they may, however, be < 0 or >= the width of the target image). As with *S3D_ProjectVector*, this is probably just good for visibility tests (maybe it can be used to determine the visibility of sections in a portal based game – that's why I added it).