1. Supplement 2016

This supplement details some topics that were extended in the course for 2016. The first (and biggest) part, bump mapping, is largely lifted from Volume 2, but this is hereby moved to TSBK07 in order to detail this important topic properly in one place, only leaving the more advanced bump mapping variant to Volume 2. After that follows a few topics that were not in the books at all before, picking and trackball controls. These two topics, unlike bump mapping, I have earlier found not important enough, but I have changed my mind. In the past, OpenGL had built-in solutions for them, which made me look past them, but now that these are deprecated that gave me a push towards looking at them more closely.

1.1 Bump mapping with extensions

In the book, we had a brief look at bump mapping, where surface detail was simulated by a bump function, which derivative was used for modulating the normal vector and thereby the shade calculated from the lighting. Alas, what you find there is a mere introduction.

In this section, we will have a closer look at bump mapping and its close relative normal mapping.

Bump mapping was first proposed by Blinn [2]. It was later modified to normal mapping by Cignioni et. al. [3].

1.1.1 Basis vectors for texture coordinates; finding the tangent and bitangent

We need a few vectors to form the coordinate system that we need, a coordinate system for the texture space. The first of these is the unmodified normal vector of the surface, **n**. We can assume that the normal vector is provided by the host program as usual, and if not we can simply use the cross product of two edges in the surface.

We need two more basis vectors, showing the direction in which the s and t texture coordinates vary. These will be located in the surface, tangents to the surface. That makes them the *tangent vector and bitangent vector.*

I will here refer to the tangent and bitangent as $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$ (basis vectors for the texture coordinates s and t). There are infinite numbers of tangent vectors to a surface, so which ones should we use? Obviously we should use the ones that correspond to the variation of the texture coordinates!

In some situations, you can create $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$ in an extremely simple way, simply by taking the cross product between the normal vector and anything, like this:

$$\hat{\boldsymbol{s}} = \frac{\hat{\boldsymbol{x}} \times \hat{\boldsymbol{n}}}{|\hat{\boldsymbol{x}} \times \hat{\boldsymbol{n}}|}$$
$$\hat{\boldsymbol{t}} = \hat{\boldsymbol{n}} \times \hat{\boldsymbol{s}}$$

This is clearly cheating, we don't care at all about the variation of the texture coordinates. The result may be horrible, but there are also cases where this works perfectly. A typical case where it works is when the bump map is just noise, with no structures.

However, for the general case we want a better method. That will be our next step.

1.1.2 Calculating the tangent and bitangent by matrix inverse

A much better method is to calculate the tangent and bitangent by some rather straight forward calculus, known as *Lengyel's Method* [1]. Let us consider a single triangle and the coordinate systems around it:

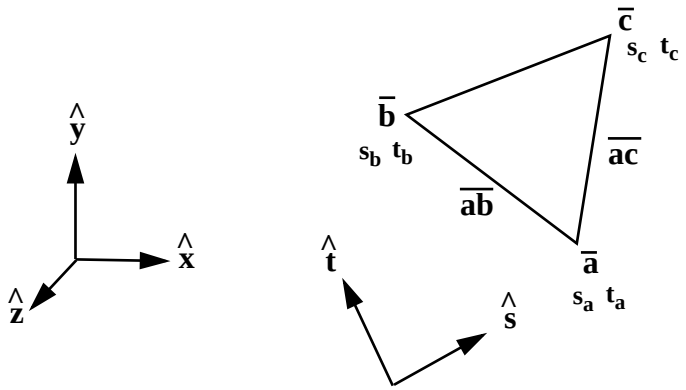

FIGURE 1. A triangle with texture coordinates and the texture coordinate basis vectors

Now we can express the edges **ab** and **ac** like this:

$$\boldsymbol{ab} = \boldsymbol{b} - \boldsymbol{a} = (s_b - s_a)\hat{\boldsymbol{s}} + (t_b - t_a)\hat{\boldsymbol{t}} = ds_1\hat{\boldsymbol{s}} + dt_1\hat{\boldsymbol{t}}$$

$$\boldsymbol{ac} = \boldsymbol{c} - \boldsymbol{a} = (s_c - s_a)\hat{\boldsymbol{s}} + (t_c - t_a)\hat{\boldsymbol{t}} = ds_2\hat{\boldsymbol{s}} + dt_2\hat{\boldsymbol{t}}$$

In Figure 2, we see how this works for the edge **ab**.

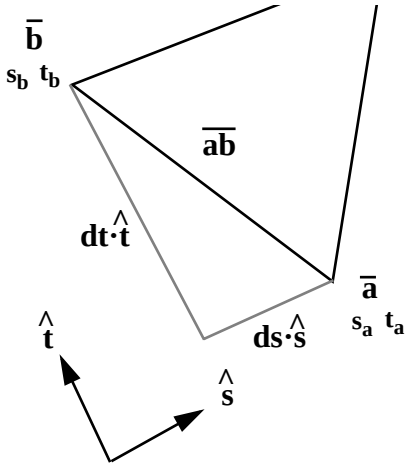

FIGURE 2. The edge ab expressed as components along s and t

We will now split the vectors to their components. Then the expressions above become as follows:

$$\begin{bmatrix} ab_x \\ ab_y \\ ab_z \end{bmatrix} = \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} ds_1 + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} dt_1$$

$$\begin{bmatrix} ac_x \\ ac_y \\ ac_z \end{bmatrix} = \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} ds_2 + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} dt_2$$

We can rewrite this as a single expression in matrix form:

$$\begin{bmatrix} ab_x & ac_x \\ ab_y & ac_y \\ ab_z & ac_z \end{bmatrix} = \begin{bmatrix} s_x & t_x \\ s_y & t_y \\ s_z & t_z \end{bmatrix} \begin{bmatrix} ds_1 & ds_2 \\ dt_1 & dt_2 \end{bmatrix}$$

Now the solution should be rather obvious: We are looking for the st matrix, and we can isolate that by multiplying both sides by the inverse of the 2x2 matrix!

$$\begin{bmatrix} ab_x & ac_x \\ ab_y & ac_y \\ ab_z & ac_z \end{bmatrix} \begin{bmatrix} ds_1 & ds_2 \\ dt_1 & dt_2 \end{bmatrix}^{-1} = \begin{bmatrix} s_x & t_x \\ s_y & t_y \\ s_z & t_z \end{bmatrix} \begin{bmatrix} ds_1 & ds_2 \\ dt_1 & dt_2 \end{bmatrix} \begin{bmatrix} ds_1 & ds_2 \\ dt_1 & dt_2 \end{bmatrix}^{-1}$$

Now we get our solution!

$$\begin{bmatrix} s_x & t_x \\ s_y & t_y \\ s_z & t_z \end{bmatrix} = \begin{bmatrix} ab_x & ac_x \\ ab_y & ac_y \\ ab_z & ac_z \end{bmatrix} \begin{bmatrix} ds_1 & ds_2 \\ dt_1 & dt_2 \end{bmatrix}^{-1} = \begin{bmatrix} ab_x & ac_x \\ ab_y & ac_y \\ ab_z & ac_z \end{bmatrix} \begin{bmatrix} dt_2 & -ds_2 \\ -dt_1 & ds_1 \end{bmatrix} \frac{1}{ds_1 dt_2 - dt_1 ds_2}$$

In C code, this will become (taken from my working demo):

```
float ds1 = sb - sa;
float ds2 = sc - sa;
float dt1 = tb - ta;
float dt2 = tc - ta;
vec3 s, t;
float r = 1/(ds1 * dt2 - dt1 * ds2);
s = ScalarMult(VectorSub(ScalarMult(ab, dt2), ScalarMult(ac, dt1)), r);
t = ScalarMult(VectorSub(ScalarMult(ac, ds1), ScalarMult(ab, ds2)), r);
```

or with C++ operator overloading:

```
float ds1 = sb - sa;
float ds2 = sc - sa;
float dt1 = tb - ta;
float dt2 = tc - ta;
vec3 s, t;
float r = 1/(ds1 * dt2 - dt1 * ds2);
s = (ab * dt2 - ac * dt1) * r;
t = (ac * ds1 - ab * ds2) * r;
```

This gives us the $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$ vectors for one particular triangle. What we should do now is to calculate them for each *vertex* instead. This is most conveniently done by finding all polygons where the vertex is a member, and take the $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$ for each of these polygons, sum together and normalize. Then the $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$ coordinate system will vary smoothly over the surface.

I find this solution quite elegant, and it does produce good results. That is, with some care in the implementation. You must check for special cases, in particular “bad” triangles where the texture coordinates don’t vary at all (making the determinant one over zero) or where an edge has zero length.

1.1.3 Approximative method for calculating the tangent and bitangent

I would like to present one more way to calculate $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{t}}$. This method is not as exact as the previous one (at least I don't have any proof for its precision) but the results are quite good.

The approach is to let each edge of a triangle contribute to the vectors, based on the direction of the edge and the variation of the s and t coordinates along the edge. Let us only consider $\hat{\boldsymbol{s}}$. For an edge **a** to **b**, **ab**, there will be the s coordinates s_a and s_b. Then that edge will add a contribution $\mathbf{s}_{ab}$ as follows:

$$\boldsymbol{s_{ab}} = \frac{\boldsymbol{ab}}{|\boldsymbol{ab}|}(s_b - s_a)$$

How can this work? Because all vertices will add to $\hat{\boldsymbol{s}}$, while their t components will cancel each other out (approximately). See Figure 3.

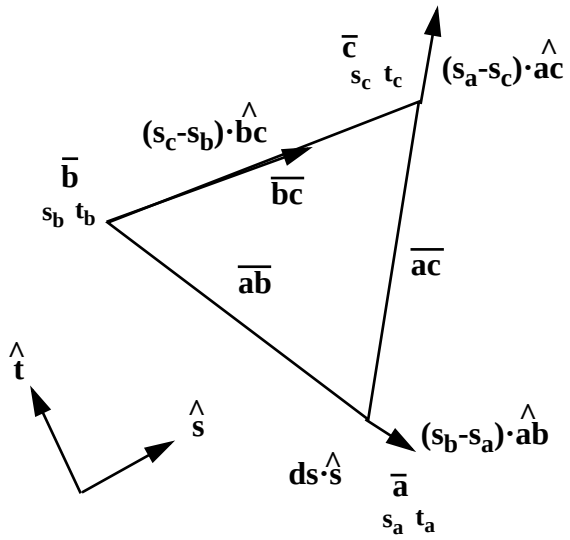

FIGURE 3. Each edge gives a contribution to an estimate of the s basis vector.

What Figure 3 tries to tell you is that, given the $\hat{\boldsymbol{s}}$ in the figure (which is what we want) the s variation is large along **bc**, and therefore it will give a large contribution. If the s difference is negative, the edge will contribute in its negative direction, which is the case for both **ab** and **ac**. We can see significant contributions in the $\hat{\boldsymbol{t}}$ direction, but they will go both ways and cancel out.

In my opinion, this method is more intuitive and may help you understand how this works. However, I would prefer the matrix inverse method since it gives an exact result.

In Figure 4 (right), you can see the Utah teapot rendered with the bump map (left image) using the method above to calculate basis vectors.

Note that $\hat{\boldsymbol{s}}$ och $\hat{\boldsymbol{t}}$ are assumed to be calculated in model coordinates. Thus, they must be transformed to view coordinates, again using gl_NormalMatrix. I will here denote the transformed, view coordinate vectors as $\mathbf{p_s}$ and $\mathbf{p_t}$.

$$\mathbf{p_s} = \text{gl\_NormalMatrix} * \hat{\boldsymbol{s}}$$

$$\mathbf{p_t} = \text{gl\_NormalMatrix} * \hat{\boldsymbol{t}}$$

1.1.4 Conversion between coordinate systems

There are no less than three coordinate systems that we must keep track of in bump mapping. Geometry is specified in *model coordinates*. We transform that to *view coordinates* to put it in the same space as light directions and viewing directions. But we also have *texture coordinates* and *tangent coordinates* (which we may here assume are the same thing).

Texture coordinates is a 3-dimensional space defined by $\mathbf{p_s}$, $\mathbf{p_t}$ and $\mathbf{n}$. We will find that there are cases where we will prefer to work directly in texture coordinates.

The transformation from model coordinates to view coordinates are, as mentioned above, done using gl_NormalMatrix. The transformation from view coordinates to texture coordinates are slightly less obvious. It is done using a rotation matrix built from $\mathbf{p_s}$, $\mathbf{p_t}$ and $\mathbf{n}$:

$$M_{vt} = \begin{bmatrix} \boldsymbol{p_s} \\ \boldsymbol{p_t} \\ \boldsymbol{n} \end{bmatrix} = \begin{bmatrix} p_{sx} & p_{sy} & p_{sz} \\ p_{tx} & p_{ty} & p_{tz} \\ n_x & n_y & n_z \end{bmatrix}$$

The matrix is trivially extended to 4x4 as needed.

Now, a few comments about the terminology. $\mathbf{p_s}$ is usually (correctly) called the *tangent vector* and is called $\mathbf{t}$ in many text, despite the obvious confusion with the t coordinate, which is along $\mathbf{p_t}$. $\mathbf{p_t}$ is often called the *binormal* (with the symbol b), which is clearly incorrect since it is rather the *bitangent*. The term binormal is most likely from a text about local coordinate systems along 1-dimensional functions, where the term is perfectly appropriate. For a surface, however, we should say bitangent. I choose to avoid the symbols t and b, using the symbols $\mathbf{p_s}$ and $\mathbf{p_t}$ instead (originally from [65] although with the names $\mathbf{p_u}$ och $\mathbf{p_v}$).

The texture coordinates, texture space, is also called *tangent space*. We may discern between the concepts *tangent space* and *texture space*, by letting tangent space be an orthonormal base while texture space has the tangent and bitangent strictly aligned with the texture directions. I choose not to consider the difference further, and rather consider

the orthonormal base to be an approximation of the strict texture space. See further Dietrich in [38]. The difference is in the direction of $\mathbf{p_t}$.

Finally, a few words about the meaning and names of the textures involved. By the term *bump map*, I refer to an image that holds height values. It could just as well be called height map, but since it is used for lighting and not geometry, I think it is reasonable to use different terms. A *normal map* is a pregenerated texture with normal vectors see below). It is sometimes called bump map, so we have some possible sources of confusion.

1.1.5 Modifying the normal vector

When we are working in view coordinates with the vectors $\mathbf{p_s}$, $\mathbf{p_t}$, $\mathbf{n}$, we can calculate the modified normal vector as

$$b_s = db/ds$$

$$b_t = db/dt$$

$$\mathbf{n'} = \mathbf{n} + b_s \cdot \mathbf{p_s} + b_t \cdot \mathbf{p_t}$$

This is the formula that was introduced in Volume 1. However, we can also consider a slightly modified definition:

$$\mathbf{n'} = \mathbf{n} - b_s \cdot \mathbf{p_s} - b_t \cdot \mathbf{p_t}$$

This is identical except for the signs. This is a pure question of definitions, namely whether the bump map height represents bumps that are out from the surface or into it. The first one is into the surface, which is arguably better since it will cause less visible artifacts than one that extends from the surface. The latter, however, is somewhat more intuitive.

This definition requires that $\mathbf{p_s}$, $\mathbf{p_t}$, $\mathbf{n}$ are orthogonal. If they are not, or not close enough to get away with it, the formula should rather be

$$\mathbf{n'} = \mathbf{n} + b_s \cdot (\mathbf{p_t} \times \mathbf{n}) + b_t \cdot (\mathbf{n} \times \mathbf{p_s})$$

After the modification, $\mathbf{n'}$ also needs to be normalized.

The calculation of b_s and b_t are done with a simple subtraction between two neighbor texels in the bump map:

$$b_s = b[s+1, t] - b[s, t]$$

$$b_t = b[s, t+1] - b[s, t]$$

This concludes bump mapping in view coordinates. But it is also possible to do this calculation in texture coordinates! Before normalization, the modified normal then becomes

$$\begin{bmatrix} b_s \\ b_t \\ 1 \end{bmatrix}$$

or (depending on the definition of the bump map direction)

$$\begin{bmatrix} -b_s \\ -b_t \\ 1 \end{bmatrix}$$

This is so simple that the more complicated formulas for view coordinates seem pretty unnecessary. However, we still need to transform light and viewing direction from view coordinates to texture coordinates. This makes the simplification a bit less exciting, but it will gain importance when using normal mapping, which is our next subject.

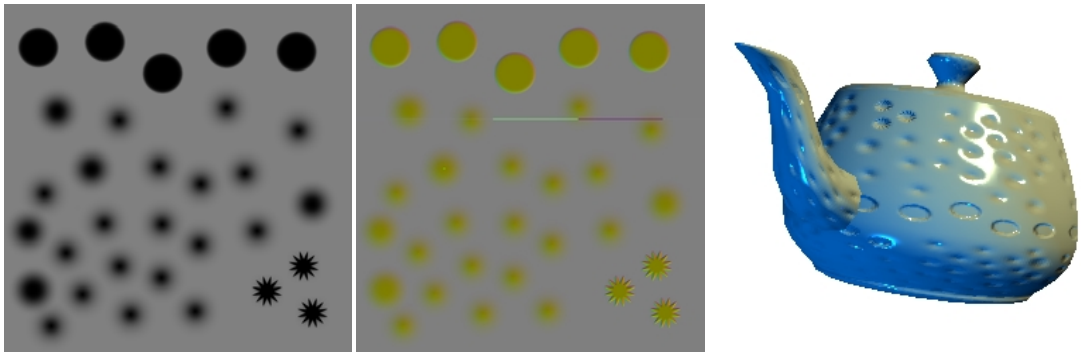

FIGURE 4. A bump map, a combined bump/normal map (most likely hard to understand if printed in black&white) and the Utah Teapot rendered using this bump map and using the texture coordinate basis vectors derived from chapter 1.1.3.

1.1.6 Normal mapping

Normal mapping is a simple precalculation of bump mapping. In practical implementations, this is the dominating method today, Thus, we should not skip it as a triviality, but instead look at its details.

The whole point with normal mapping is that we precalculate the normal vector, in texture coordinates. Above, we found that it can be calculated as

$$\begin{bmatrix} -b_s \\ -b_t \\ 1 \end{bmatrix}$$

(or, again depending on definition, without the negations). This is calculated straight from the bump map image data, as

$$-b_s = b[s, t] - b[s+1, t]$$

$$-b_t = b[s, t] - b[s, t+1]$$

1

It is then normalized, which means that each component is within the interval [-1..1]. It should then be placed into a texture. To store it in a texture, we need to scale and bias the data, since textures can only hold the interval [0..1]:

$$R = (x+1)/2 \qquad G = (y+1)/2 \qquad B = (z+1)/2$$

After this transformation, we store the data in the red, green and blue channels of an image. This procedure creates a normal map from a bump map as shown in the figure below (images from an example found on the web).

FIGURE 5. A bump map commonly used in demos (left), and its normal map (right).

In Figure 4 (middle), you can see a normal map combined with the bump map. That makes the result somewhat hard to watch but the combined data can be useful.

When rendering, the normal vectors are recovered by

$$x = 2R-1 \qquad y = 2G -1 \qquad z = 2B -1$$

which gives us the normal vector in texture coordinates as $\mathbf{n}^t = (x, y, z)$.

The light direction is transformed to texture coordinates

$$\mathbf{L_t} = M_{vt} \cdot \mathbf{L}$$

and likewise for the viewing direction. Thereby, we have normal vector, light direction and viewing direction all in the same coordinate system, which is sufficient to make lighting calculation as desired.

Thus, normal mapping requires some extra calculations, but also save on two important points: We save the normalization of the modified normal vector, and we reduce the texture accesses by 50%.

1.2 Applications of ray-casting

Ray-casting may sound like an insignificant first step of ray-tracing, but that is not the case. It is not just a visible surface detection method, but has several other applications, including the following:

- Picking
- Fairly efficient rendering of 3D scenes defined by a 2D (Wolfenstein 3D style) or 3D grid (Minecraft style world), or a 2D heightmap (Comanche).
- Searching a scene for purposes of illumination.
- Volume visualization of transparent object, where you cast rays through objects, accumulating their density. This is another case of rendering with a 3D grid.
- Visualization computation, typically for game AI. In this case, a 3D scene is typically reduced to a simplified 2D representation, and the problem is reduced to a 2D grid ray-casting problem.

Considering the grid rendering, I must note that I do not know exactly how specific games are rendered, like Minecraft, but I do know that there are engines for voxel worlds based on ray-casting, and I have written a Wolfenstein-style 2D grid raycaster myself, running smoothly on a 33 MHz computer.

1.3 Picking

Picking is a common problem. It is the seemingly simple problem of selecting an object with a mouse. In a 2D scene this is usually trivial (possibly with some complications with complex shapes like splines), but in a 3D scene it is quite different, and there are several solutions to choose from. These include:

- Ray-casting in view coordinates
- Ray-casting in model coordinates
- Indexing models by color

Out of these, the first may seem like the most natural choice, the second is more efficient, and the third easiest to implement but least flexible in usage.

1.3.1 Picking by ray-casting

Ray-casting, as described in “Polygons Feel No Pain”, is the task of tracking a ray from the camera into a scene. We do it here almost the same way. Start from the camera, the projection reference point, through one pixel, into the scene. For picking, only onne ray is cast, and it goes though the point where the user clicked the mouse.

However, this assumes that all models are transformed to view coordinates! See Figure 7. If we do that, we get *picking in view coordinates*. Doing all those transformations on the CPU can require considerable computations unless considerably optimized. However, it is also possible to do this on the GPU as part of the rendering. This would imply that we perform picking in the geometry stage. Then the actual picking is easy but it is less obvious how to pass the result to the CPU. (You can, for instance, use the write-to-texture feature of newer GPUs.)

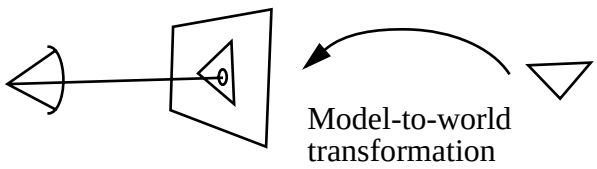

FIGURE 6. Ray-casting in view coordinates

Another option is to perform picking in model coordinates! Then you transform the ray by the inverse model-to-view transform for *each model*. Since the models tend to be complex, this will be significantly faster. See Figure 7. Of course, we still have to make many ray-in-triangle checks so we have only optimized part of the problem.

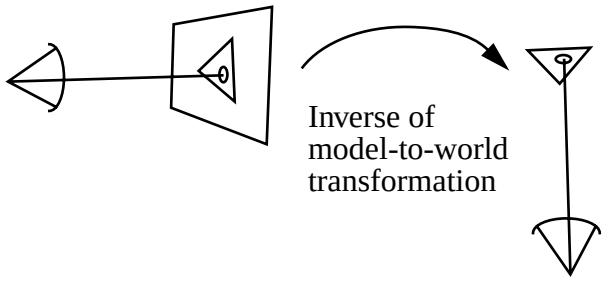

FIGURE 7. Ray-casting in model coordinates

Picking by ray-casting will give you an exact point within a specific triangle in a model, not only identification of a model. This gives you great possibilities in doing more than just identifying an object in the scene.

Considering how to optimize the raycasting in the models, I refer to the raytracing chapter. Basically, the usual principles apply with bounding shapes, and subdivisions. My first step would be a bounding sphere.

1.3.2 Picking by color index

A completely different way to perform picking, which is notably straight-forward, is to assign every separate model a unique color, a color that has no coupling to its look but is simply an identifier. The scene is then drawn with each model rendered in a single color, no shading or textures, like in Figure 8.

FIGURE 8. Three teapots rendered with distinctly different colors, making picking easy

This method is known as “The WYSIWYG method”, and was, according to other publications, by Robin Forrest in the mid-1980’s and used by Hanrahan and Haeberli for a 3D painting application [4].

Assuming that we render to a standard RGBA buffer with 8 bits per color, we have a maximum number of indices of 2^{32} = 4.3 billions (including alpha), or 16 millions if we skip the alpha channel. However, I would advise a larger distance between the colors in order not to get accidents by interpolated values at edges. This will depend on the application, but in many cases you will be happy with a handfull of objects.

Once the image has been rendered, you can pick up the pixel at the mouse click by a call to glReadPixels. This call will get all pixels in a rectangle, so we choose a 1x1 pixel rectangle. Then we go through our list of color indices and see if we can find one that is within a small tolerance.

Now, you probably don’t want the user to see your models drawn in flat colors like this. No problem, this can be avoided. You just draw the scene twice, once with indices and once with the ordinary look. You don’t even have to render the color index image before the user clicks the mouse!

There are (at least) two options for doing this. Either you render to the ordinary image buffer, but without swapping buffers, pick up the color at the mouse pointer, and then erase the scene and render the image that the user should see, or you render the index color image off-screen, to a Framebuffer Object (i.e. render to a texture).

1.3.3 Old-style picking

It should also be mentioned that OpenGL had a built-in picking feature in the past. This is what you will usually find if you Google on the topic. However, that method is now obsolete and the methods described above are recommended.

1.4 Trackball-style controls

Trackball-style controls mean that we click-and-drag the mouse on an object in a scene in order to rotate it. This fits well with picking for manipulating one object out of several, or, as a simpler case, it can mean that we have a single object in the scene that is used for illlustrating some effect.

This is a rather straight-forward application of rotation around an arbitrary axis, and just like in many other cases, the big thing is to keep track of the coordinate systems.

The user input, a click-and-drag of the mouse, gives us a movement on the screen, which translates to a vector in view coordinates. This should then be translated to rotation of an object (an instance of a model placed in the scene). However, this object may be rotated and translated, and the camera too. Thus, the transform should be applied in some waythat the object is rotated in place, but not translated. The object rotation should be affected, but not the translation; we must apply a rotation so that no existing translation is rotated!

If, following my illustration of the transformation chain, the world-to-view transform is R_v*T_v, and the model-to-world is T_w*R_w (now, again, why do I use this order?), then the rotal model-to-view transformation is $R_v*T_v * T_w*R_w$. Now the rotation vector should cause a rotation somewhere in this chain.

And here is my solution: The mouse movement is a vector $\mathbf{v} = (v_x, v_y, 0)$. From this, I can find the orthogonal vector $\mathbf{u} = (u_x, u_y, 0) = (-v_y, v_x, 0)$. This is now transformed to world coordinates as $R_v^{-1}\mathbf{u}$. Then I use this vector as axis to rotate around, with its length for giving the rotation angle. This resulting rotation matrix is then inserted *between T_w and R_w*. This means that any existing model-to-world rotation is unaffcted, while any existing rotation is added upon by the new rotation matrix.

This approach is illutrated in Figure 9, As so often, you can reformulate the problem in other ways and work in other coordinate systems, but I find this solution reasonably straight-forward.

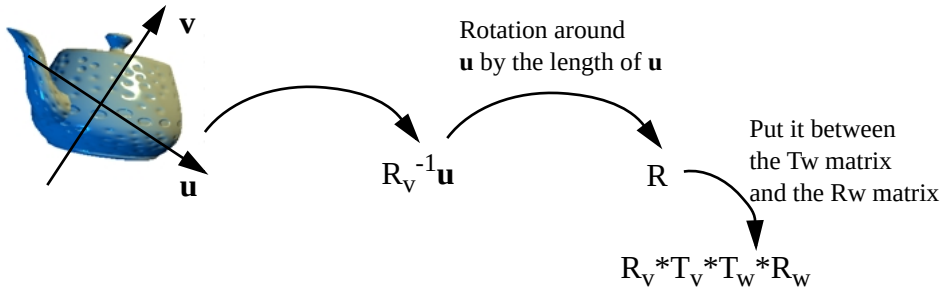

FIGURE 9. A way to implement trackball-style rotation from a mouse movement by v.

This simple problem is sometimes considered to hard that you need reusable code for it, but was it really that hard?

1.5 Particle systems and instancing

(Extended version)

Particle systems have been mentioned (section 13.9.7 on page 176) as a typical application for billboards. That is, billboards are often use to draw the particles in a particle system. An equally important problem is how to move the particles.

A particle system is an entity in a graphics system where a fairly large amount of point-shaped objects, particles, are handled. Typically, all the particles in a specific particle system have similar behavior. Quite often, only the initial values (position and speed) vary.

A particle can be represented by a small data structure with fields for position and speed. Fields for acceleration, mass, physical size (radius) and look etc may also be included. (See further below.)

On these particles, the simplest laws of physics are applied:

$$\text{acceleration} = \text{gravity} + \text{forces/mass}$$

$$\text{speed} = \text{speed} + \text{acceleration} \cdot \Delta t$$

$$\text{position} = \text{position} + \text{speed} \cdot \Delta t$$

In many cases, gravity is the only force that applies, in which case the particle will move in a typical missile trajectory parabola.

If these equations are simply updated once for every frame (preferably with some factors for the time step) you get *Euler integration* of the speed and position. This is a crude approximation of the true integral, but usually quite sufficient for particle systems.

The life-span for a particle can be very short. If the particle system simulates a fountain (a very good application for particle systems) the particle is only interesting until it hits the ground or water surface.

Particle systems come in many kinds. We can categorise them in three groups:

- Independent particles
- Dependent, freely moving particles
- Connected particles

The independent particles are by far t he easiest to create. They only need position and velocity, and require no other tests than a test for the termination. These particle systems are suitable for effects like snow, bursts of water (fountains, waterfalls), clouds and smoke.

Dependent particles, particles for which we perform collision tests with no prior knowledge of which particles may be close, are by far the hardest to do, performance wise. A trivial implementation will have very large complexity, $O(N^2)$.

Finally, connected particles are moderately easy to build and easier to manage (lower complexity), but the hardest to make stable. A typical case here is to simulate cloth. Euler intergration is not sufficient here and may make the particle system unstable.

For our purposes, for this book (part 1) we primarily consider independent particles. More advanced particle systems implemented on the GPU are subjects for part 2.

There are multiple ways to use a particle system to create geometry, but the easiest way to draw a particle system is to draw the individual particles as billboards. This is easy for very small particle systems, but when the numbers rise, the number of function calls becomes so large that it becomes the bottleneck. For this we may use a technique called *instancing*.

What instancing does is simple enough: It draws a specific shape several times at once, with a single function call! This is done with the calls glDrawArraysInstanced or glDrawElementsInstanced, variants of the usual calls adding a parameter which specifies how many instances that should be drawn. This would, of course, draw all instances on top of each other if there was no way to handle them separately. This is done using the variable gl_InstanceID in the shaders, primarily in the vertex shader to affect the location of the instance.

As example, here follows the vertex shader from our demo “Billboard instancing” (available in the PFNP demo archive).

```
#version 150

in  vec3 in_Position;
uniform mat4 myMatrix;
uniform float angle;
uniform float slope;
out vec2 texCoord;

void main(void)
{
   mat4 r;
   float a = angle + gl_InstanceID * 0.5;
   float rr = 1.0 - slope * gl_InstanceID * 0.01;
   r[0] = rr*vec4(cos(a), -sin(a), 0, 0);
   r[1] = rr*vec4(sin(a), cos(a), 0, 0);
   r[2] = vec4(0, 0, 1, 0);
   r[3] = vec4(0, 0, 0, 1);
   texCoord.s = in_Position.x+0.5;
   texCoord.t = in_Position.y+0.5;
   gl_Position = r * myMatrix *  vec4(in_Position, 1.0);
}
```

The important part here is, of course, the use of `gl_InstanceID` to place the instances in different places. The uniforms angle and slope are used to fine tune the result, which may look like Figure 10.

FIGURE 10. Demo of instancing of billboards.

This simple program lends itself to considerable creativity, but also has uts problems. If we want the particles to move around with some noise added for variation, how can we do that? How do we get some more data into the shaders? How about a texture with noise?

It should be noted that instancing is at its best for simple models like a billboard. For more complex models, the number if function calls may no longer be the bottleneck, and then instancing is not as significant.

1.6 Just a little patch

The three-component light calculation should be like this:

$$i = k_d i_a + \sum (k_d \cdot i_s \cdot \max(0, \cos\theta_s) + k_{spec} i_s \cdot \max(0, \cos\phi_s)^n)$$

2. References

References, references... *never* forget the references!

[1] Eric Lengyel, “Mathematics for 3D Game Programming and Computer Graphics”, 3rd edition, 2011.

[2] J. Blinn, “Simulation of wrinkled surfaces“, in *Proceedings of the 5th annual conference on Comåputer Graphics and interactive techniques*, 1978

[3] P. Cignioni, C. Montani, C Roccini, R. Scopigno, “A general method for preserving attribute values on simplified meshes”, *Proceedings, IEEE Visualization*, 1998

[4] P. Hanrahan, P. Haeberli, “Direct WYSIWYG Painting and Texturing on 3D Shapes”, Computer Graphics, Volume 24, Number 4, 1990.