The University of Adelaide, School of Computer Science
Computer Graphics,
Semester 1 2014
Tutorial 5 – Shaders – Bump Map and Parallax Map

This tutorial is based on code recently placed on the website from the wikibooks tutorials. Specifically the "textures_in_3d" example. You will need a couple of new texture files – these have been placed on the course website.

Read the explanation of how the original code implements Bump Mapping at http://en.wikibooks.org/wiki/GLSL_Programming/GLUT/Lighting_of_Bumpy_Surfaces

Check you understand how the code implements this strategy.

After you are sure you understand the code – run it and look at the bump mapping effect.

Questions/Thoughts – (not to detract from the "niceness" of the idea) – it is clear that this is "fudging" actual surface variation in heights. What cues are there that it is playing with the lighting through "surface normals" (quotes used because the surface is in reality still flat but we've artificially varied the normal) rather than actually perturbing the heights? What about the edges of the cube? – these are still pretty well perfectly straight. But also, though the normals/lighting gives the impression of significant variations in surface height, it's wrong in a subtle way as there is no parallax (parts perturbed to be higher should actually shift in their position seen by the viewer). We can't (easily) actually do this since the vertex shader works only on vertices, and, by the time the flow has reached the fragment shader, the positions in screen coords has been determined. But we can look up a shifted normal. This is the fudged parallax effect we will implement.

You might find it useful to read a tutorial that implements both Bump and Parallax Mapping (we are going to add the latter to the wikibooks code):
http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_4.pdf
Of course, although this tutorial implements bump mapping as does the wikibooks, there are differences in the way it is done and (large) differences in the driver code/interface (the wikibooks, for examples fixes the light properties in the shader rather than having them passed from the application program – for example). Also, you have to be consistent in applying calculations in the same coordinate system. The varying normals of a normal texture map are in **local** (defined by the polygon tangents and normal) coordinate system; whereas, the viewing vector, light vectors etc are in some **global** coordinate system (world coordinates or eye coordinates typically). It doesn't make sense to combine (add and subtract vectors, dot product etc) expressed in different coordinate systems – so you have two choices (at least) convert all to local (and carry out the calculations here) or convert all to global (and carry out the calculations here).

You might also look at:
http://en.wikibooks.org/wiki/Cg_Programming/Unity/Projection_of_Bumpy_Surfaces
which also explains these concepts. In particular, the figure there is a lot more simple for explaining the concept of parallax mapping. In short, we need only calculate the shift vector by getting the view direction (in local coordinates), projecting it onto the local tangent plane, and then scaling by the height. (You don't need to fight your way through the geometry if you don't want because it is rather obvious that the shift is in the direction of the projection of the view direction onto the (unperturbed) surface and than the shift will be zero if the height displacement is zero and will be greater as the height gets greater. And, after all, we only need an effect that "looks right" by varying in the right sort of manner – we usually tweak the various factors for the amount of shift we want.

Your task is to modify the wikibooks code to implement the same "switching behavior" of the TyphoonLabs code – that is, have the user by mouse clicks or key presses, select between:

a) No bump and no parallax
b) Bump but no parallax (this is your starting code)
c) Both bump, and parallax

Hints and Steps:

1. We are not only going to perturb the normal (using a normalmap texture) but to perturb the height (and use the new height to calculate an approximate offset to the new surface normal offset). Thus we need two textures rather than one (multitexturing). You have already been pointed to example code that uses multi-texturing in the same wikibooks tutorial bundle. The texture mapping of the world:
   http://en.wikibooks.org/wiki/GLSL_Programming/GLUT/Layers_of_Textures "basic texturing"
   uses a night texture and day texture – you need to assign one texture to texture unit 0 and one to texture unit 2 so you can have both textures accessible at the same time.
   Since, for this exercise, you need both a normal and a height texture map – but the example code has only a normalmap. Hence, we are going to switch from the Brick normal texture (of the starting code – we don't have the corresponding height image) to the supplied rock height and normal images. So your first task – other than "getting your head around" multi-texturing and parallax mapping, is to do a trivial modification to read in the normal map from the rock image.
   2. Your next change is also rather simple – it is to create version a) above (in addition to the version b) that we have from the starting code – user switchable between the two version.
   3. Finally, tackle the parallax mapping. We need to read in the height map and use it (so modify your code – hint: since we want opengl to know this is a greyscale image/texture, when you read in the texture by SOIL use the `SOIL_LOAD_L mask bitwise OR'd in the last parameter set to the load function`).

Make sure the added functionality is a user switchable addition so you can rapidly switch back and forth to compare with and without parallax mapping. This is a big step – I suggest you work incrementally on these elements: loading the height texture and modifying the shader to perform the calculations.
As stated before, we have to be able to transfer from local coords (the normal is encoded as a vector relative to the local surface coords of the polygon). The local surface coordinate frame is defined by the directions of the surface/polygon normal (our texture is to be read as a replacement for that), "the" tangent vector (it's one of the infinite number of tangent directions) and the so called b vector that is the cross-product between the tangent vector and the normal. New coordinates are nothing more than projections of a vector onto the new axes of that coordinate frame. That is, we convert to world coordinates of some vector e by $v = ((M_n t).e, b.e, (M_n n).e)$ where $b = (M_n n) \times (M_n t)$, where M_n is the transformation matrix for normals, n is the normal vector and t is the tangent vector. This equation can be written in Matrix vector multiplication with a matrix whose rows are $(M_n t)$, b and $(M_n n)$. The code you already have computes this matrix **in the vertex shader** so that the entities (rows) are interpolated as passed down to the fragment shader – find it in the code! You might find you need the inverse transformation – with more recent glsl's you can invert the matrix direction in the shader (whether this is efficient or not is another issue) but also note that the inverse of a unitary matrix (as are all coordinate transforms) is the transpose of that matrix. You might invert/transpose in the vertex shader – or in the application and pass it through the vertex shader - there are several possibilities.

In the fragment shader, we will take the height map and use it to change the texture coordinates based on the height of the current texture position and the eye position. Assume that the height at t_o (the original texture coordinate) is h. It is necessary to scale and bias h because its is in the range [0, 1], as follows: $\tilde{h} = hs + b$. (s and b are factors you can "play with"). Finally the texture offset for point P is then computed by projecting a vector from the point $\tilde{h}$ above P to the eye, onto the polygon. Projecting onto the polygon is nothing more than taking the first two coordinates of vector when transformed into the "right" coordinate system - calculated by the equation above (you can see this because these two coordinates are in the transformed directions of the two surface tangent vectors) – i.e., v.xy. You then scale by $\tilde{h}$. This new vector is the offset and can be added to the original texture coordinate to produce the new texture coordinate t_n, as in $t_n = t_o + \tilde{h}$ v.xy.