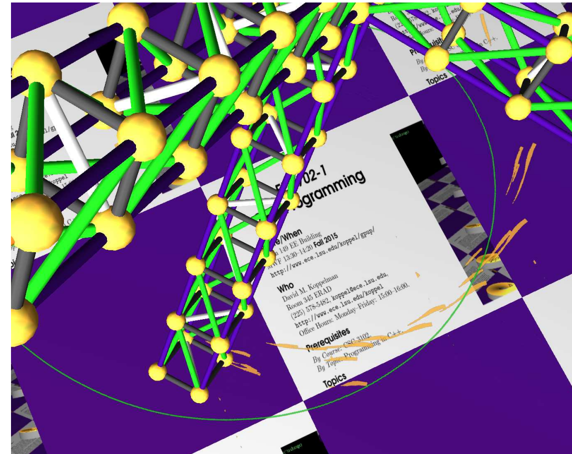
**LSU EE 4702-1** **Homework 3** Solution **Due: 12 October 2015**




**Problem 0:** Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should show a swinging string of beads. Press digits `1` through `3` to initialize different scenes, the program starts with scene 1. The illustration to the right is from scene 3 generated with a solution to Problem 1. *Promptly report any problems.*

The code in this package includes the friction model asked for in Homework 2. The links however are unbreakable and there's no skin. Also, shadows are deactivated by default to make the solution to this assignment easier. Notice that this starts like the code for Homework 2, except that ball at the bottom of the chain hits the platform as it swings by.

Use the arrow keys, PageUp, and PageDown to move around the scene. Use key `h` to toggle between the first (head) ball being locked in place and free. Use key `t` to do the same for the last (tail) ball. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye.

*Note: There is nothing to turn in for this first problem.*

The solution code for this assignment has been checked into the repository in file `hw03-sol.cc`. For an HTML version visit `http://www.ece.lsu.edu/koppel/gpup/2015/hw03-sol.cc.html`. The repository version will always be most up to date.

**Problem 1:** Notice how the platform is always perfectly clean. That's not realistic! In this problem modify the code so that when the balls slide across the platform they leave scuffs. The shape of a scuff should be the area of the platform touched by the ball (base the width on the intersection of the sphere and the platform) and the color should match the ball. See the illustration to the right. The ball at the base of the gyroscope is green, it leaves a neat green path. The arm balls are yellow. The balls at the arm ends striking the platform leave scuffs that start out thick and get thinner.

The code should be added to class `My_Piece_Of_The_World` and elsewhere as needed. The places in which code needs to be added start with the comment `Homework 3` and are followed by checklists of things to be done.

Scuffs are to be implemented using textures. Rather than using one large texture covering the entire platform, the textures are broken up into a grid of *overlays*. The texture corresponding to an overlay should only be initialized when the area is first scuffed, and textures should only be sent to OpenGL when the textures change. This uses vastly less data then sending a texture for the entire platform at every frame.

(*a*) Code has already been added to `time_step_cpu` to retrieve an overlay, actually a `Platform_Overlay` object, for a ball if it is colliding with the platform. Add code to this routine and elsewhere to write the appropriate texels with scuff marks. For sample code see `sample_tex_make` in the homework file. This routine creates a texture containing a big red ex. (The code in `time_step_cpu` only needs to modify the texels in `po->data`, it should not send the data to OpenGL!)

The key to solving this problem is understanding how to go from object space coordinates, which are of course used for ball positions, to a texel coordinate space, and how to use a texel coordinate to compute an index for the texel array.

The routine `po_get_lcoor` converts from object space to texel space:

```
  // The two lines below are from My_Piece_Of_The_World::init()
  scale_x_obj_to_texel = wid_x_inv * twid_x;
  scale_z_obj_to_texel = wid_z_inv * twid_z;

pCoor My_Piece_Of_The_World::po_get_lcoor(Platform_Overlay *po, pCoor pos) {
  pCoor lc;
  lc.x = ( pos.x - overlay_xmin ) * scale_x_obj_to_texel;
  lc.z = ( pos.z - overlay_zmin ) * scale_z_obj_to_texel;
  lc.y = 0;
  lc.w = 0;
  return lc;
}
```

Variables `overlay_xmin` and `overlay_zmin` are the object space coordinates of a corner of the overlay, these were set in another routine when the overlay was retrieved. The `x` and `z` components of the returned coordinate give texel locations corresponding to `pos`. These will be used to access our texel array, in `data`, which is size `twid_x*twid_z` elements. The index into that array is computed by

```
int My_Piece_Of_The_World::po_get_tidx(pCoor lpos) {
  // Return an index for the texel array corresponding to texel coord lpos.
  const int idx = int(lpos.x) + twid_x * int(lpos.z);
  return idx;}
```

The next step is to find the object coordinates of the area to be scuffed. If we just wanted to scuff a single texel we could call `po_get_lcoor` and `po_get_tidx` to get the index into the array and then set `po->data[idx] = ball->color`, where `idx` is the return value of `po_get_tidx`. But, we want the size of the scuff mark to be based on the force, and we also need to account for motion between time steps. The code below handles both. Notice that the `t/u` loop nest is essentially rasterizing the rectangle corresponding to the scuffed area.

```
      pCoor ball_lcor = mp.po_get_lcoor(po,ball->position);

      // Previous location of ball in texel coordinates.
      pCoor prev_lcor = mp.po_get_lcoor(po,pos_prev);

      float width = mp.scale_x_obj_to_texel *
        sqrt( ball->radius * ball->radius - pos_prev.y * pos_prev.y );

      // Direction along scuff mark (based on sliding motion).
      pNorm skid(prev_lcor,ball_lcor);

      // Find direction along width of scuff mark.
      pNorm nskid = cross( skid, pVect(0,1,0) );
```

```
    // Iterate over texel locations scuffed by ball.
    for ( float t = -width; t <= skid.magnitude+width; t++ )
      for ( float u = -width; u < max(width,1.0f); u++ )
      {
        pCoor tex_pos = prev_lcor + t * skid + u * nskid;
        pColor* const texel = mp.po_get_texel(po,tex_pos);

        // If texel is not on overlay po then just skip it.
        if ( !texel ) continue;

        // Don't bother if already scuffed. If we wanted to be
        // fancy we could add this balls color onto what is already
        // there. But we don't want to be fancy.
        if ( texel->a ) continue;

        po->texture_modified = true;
        *texel = ball->color;
        texel->a = 0.8;
      }
```

The solution above does not touch an already scuffed region. It would not be too difficult to modify the code so that a second scuff combines colors with any existing scuff.

(*b*) Routine `My_Piece_Of_The_World::render()` is supposed to render the platform overlays. For each scuffed overlay it should emit primitive(s) with the corresponding texture attached. The texture application, alpha test, and blending modes need to be set so that only the scuff marks cover the underlying platform (and semi-transparently if possible). The routine has some code. Finish it, and make changes elsewhere as needed.

See `gpup/demo-8-textures.cc` for examples of how to do texturing, blending, and alpha testing. If you are not sure if your own texture is working you can use the texture created in `sample_tex_make`.

There are several things that need to be done. At the top of the render routine we need to set up texturing and blending. Then for each non-empty overlay we need to do the following: First, if the overlay is new we need to create the buffer object. Second, if our copy of the texture (the one in `data`) has been modified, we need to send it to OpenGL. Next, we need to render a primitive on which the texture will be applied.

To start, the lines below turn on texturing and select the modulate texture application mode. The application mode indicates how the texel color will be combined with the primitive color (actually, the color from the prior texture unit, but in this case there is no prior texture unit). The solution uses modulation, in which the color values are multiplied. My multiplying lighting effects are transferred to the texel values.

```
glEnable(GL_TEXTURE_2D);
glActiveTexture(GL_TEXTURE0);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

The solution uses two methods of applying the textures to the frame buffer. The primary one is through blending, in which the stuff mark and the existing pixel values are combined. That's set up by turning on blending and setting blending to use the texel's alpha value, and to not blend the alpha values themselves.

```
glEnable(GL_BLEND);
glBlendFuncSeparate(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA, GL_ZERO, GL_ONE);
```

Inside of the overlay (`i`) loop, a new texture object is initialized if necessary. This should be done once per overlay.

```
for ( int i=0; i<num_overlays; i++ ) {
```

```
Platform_Overlay* const po = &platform_overlays[i];
if ( !po->data ) continue;

if ( !po->texture_object_initialized ) {
    po->texture_object_initialized = true;
    glGenTextures(1,&po->txid);
    glBindTexture(GL_TEXTURE_2D,po->txid);
    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, 1);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  }
```

Next, texture data for this overlay is sent to OpenGL *if the texture changed.*

```
glBindTexture(GL_TEXTURE_2D,po->txid);
if ( po->texture_modified ) {
    po->texture_modified = false;
    glTexImage2D
      (GL_TEXTURE_2D,
       0,                // Level of Detail (0 is base).
       GL_RGBA,          // Internal format to be used for texture.
       twid_x, twid_z,
       0,                // Border
       GL_RGBA,     // GL_BGRA: Format of data read by this call.
       GL_FLOAT,    // Size of component.
       (void*)po->data); }
```

Next a primitive to carry the texture is rendered. The primitive is a quad (quadrilateral), the coordinates were pre-computed when the texture overlay was created.

```
glBegin(GL_QUADS);
glNormal3f(0,-1,0);
glColor3fv(color_white);
glTexCoord2f(0,0);
glVertex3fv(po->vertices[0]);
glTexCoord2f(1,0);
glVertex3fv(po->vertices[1]);
glTexCoord2f(1,1);
glVertex3fv(po->vertices[2]);
glTexCoord2f(0,1);
glVertex3fv(po->vertices[3]);
glEnd();
```

Routine `po_get` in the solution computes the coordinates of the overlay boundaries used above, those are put in array `vertices`. The routine also allocates and initializes the texel array.

The routine `clean` removes scuff marks. It does so by freeing our texel arrays (`data`) and OpenGL's texture objects, making the storage available for other use.

A common mistake was to implement clean with the single line `platform_overlay = new Platform_Overlay[num_overlay` That would keep the CPU storage for the textures (in `data`) and the GPU storage (in the texture objects) in use, a situation that is called a memory leak. Since the textures can be large, the memory leak would be serious.

**Problem 2:** Answer each question below about performance aspects of texturing.

($a$) The texel array that we send to OpenGL is an array of `floats`, because that lets us use our familiar `pColor` objects. What would be the impact of using a color representation that matched the depth of our displays, eight bits per color? *Note: This question touches on more issues than were anticipated, and it would not be easy to give a fully correct answer given the material covered in the course.*

Relatively Short Answer: Assuming that the texel is sent to the GPU in the format provided to OpenGL, the use of an 8-bit integer rather than a 32-bit float would reduce the amount of data sent to the GPU by a factor of four, and since the texture arrays we are working with are of moderate size and change frequently the savings would be significant.

Long Answer: The impact that the question asks about can take three forms: the amount of computation needed to convert the texel colors to OpenGL's internal format, the amount of data that needs to be sent from the CPU to the GPU, and the range of representable colors. The OpenGL standard describes behavior in terms of a floating-point representation for colors, and it is likely that modern GPUs use such representations. (Though they may not be 32-bit IEEE 754 floats.) If the representation were 32-bit IEEE floats, then additional computation would be needed if the colors were given in integers. If the conversion were performed on the GPU, then providing the data as integers would reduce the amount of data to be sent by a factor of four, which is a large amount. The actual computation needed to convert to the internal format would be small compared to the computation needed to prepare the different MIPMAP levels. We can assume that a decent OpenGL would do this on the GPU (though the fact that automatic MIPMAP generation is now deprecated might make this less likely in the future).

($b$) Suppose we vary `nx` and `nz`, and choose `twid_x` and `twid_z` so that the total number of texels across the platform, `nx * twid_x`, is constant. Describe the impact on performance as `nx` and `nz` change vary from 1 to their maximum (say, 10240 for the assignment code).

When `nx` and `nz` are small the textures are huge, and so performance would be hurt when we need to send them over. When `nx` and `nz` are large the textures are small but there are many more textures and especially many more `Platform_Overlay` objects. Though the amount of texture data that is sent to the GPU is smaller with large `nx` and `nz` (since we only send modified textures), the overhead of managing these textures becomes large. At their maximum value each texture is just one texel, that's 16 bytes, but that's comparable to the size of the `Platform_Overlay` object needed to manage them. While the storage for textures is only created when the respective area is scuffed, the `platform_overlays` array is sized to `nx * nz` elements, which would be $10240 \times 10240 = 104\,857\,600$ elements taking over a GB of storage even for a perfectly clean platform. Also, the rendering code performs an exhaustive search of overlays to find the dirty ones. If that code isn't modified there would be a performance hit there too. Even if the `Platform_Overlay` objects were themselves only allocated for scuffed textures, performance would still be hurt because the time needed to prepare to render the overlay would have to be born for each little one-texel texture. If the scuffed region covered 500 texels, that would be 500 times. If `nx` were smaller (and textures larger, say 50 by 50 texels) then perhaps 2 or 3 overlays would be sufficient to cover the scuffed areas, and so the rendering preparation would only be done 2 or 3 times. All off this means that intermediate values of `nx` and `nz` yield the best performance.

Just to be sure that everyone understands, lets start with `nx` and `nz` at 1 and increase them to their maximum value. When they are at 1, we need to send over a $10240 \times 10240 \times 4 \times 4 = 1600\,\text{MiB}$ texture each time the texture changes, which could be every frame. If we increase `nx` and `nz` to 2, now there are four overlays. If new scuff marks are confined to one of those four we only need to send over $400\,\text{MiB}$ of data, which should yield much better performance. Increasing them to 16 means we have $6.25\,\text{MiB}$ textures and $16^2 = 256$ overlays, which ought to improve performance even more. But at some point, maybe when `nx` and `nz` reach 1024 the size of the `platform_overlay` array will start to hurt performance, since we scan the entire thing before rendering. Increasing them further will cause our application to suffer due to the time needed to render all those tiny textures.