Exam – Solutions
EDA221 Computer Graphics : Introduction to 3D

2013–10–21

Note that this document only shows suggested solutions, and may not represent the *exact* solutions needed to get full score on the exam.

1. Shading
   a) Diffuse and specular terms of Phong shading

$$I = \sum_i \left(\underbrace{k_a L_{i_a}}_{\text{ambient}} + \underbrace{k_d \max(0, \mathbf{l}_i \cdot \mathbf{n}) L_{i_d}}_{\text{diffuse}} + \underbrace{k_s \max(0, (\mathbf{r}_i \cdot \mathbf{v})^\alpha) L_{i_s}}_{\text{specular}} \right)$$

   The Phong model consists of an ambient, diffuse and specular term. Replacing the specular term with Blinn-Phong specular $(\mathbf{n} \cdot \mathbf{h})^\alpha$ is also an accepted solution. For full score, define the diffuse and specular terms, their coefficients and the included vectors. Also include the max operators. Notice that there can be several light sources and that the vectors need to be normalized.

   b) Yellow, mostly diffuse material with a small specular peak: $k_a = (0, 0, 0)$ (this term approximates indirect lighting and should be set low or zero, in any case, it should be considerably less than the diffuse term), $k_d = (.7, .7, 0)$ and $k_s = (0.3, 0.3, 0.3)$. Finally, the shininess (α above) should be set to a large value, say 100, to get a small specular peak.

   c) Color at middle of the edge: Note that the graphics pipeline automatically interpolate all vertex attributes for each pixel, so the color will vary from $(1, 0, 0)$ at $\mathbf{p}_0$ to $(0, 1, 0)$ at $\mathbf{p}_1$. At the middle of the edge, the color will be $(0.5, 0.5, 0)$ The vertex shader simply passes on the normal values, and the normal is used to set the pixel color.

   d) Write a vertex shader that, when applied to a sphere model, makes the sphere grow and shrink

```
in vec4 vPos;
in vec3 vNormal;
uniform float time;
uniform mat4 MVP;
void main()
{
    float scale = 1.0 + sin(time);
    vec3   pnew = vPos.xyz + vNormal*scale;
    gl_Position = MVP*vec4(pnew, 1.0);
}
```

2. Transforms

   a) Given the point $P = (1, 2, 3, 1)$, what is the location of $P' = \mathbf{ABC}P$?

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$\mathbf{B} = \begin{bmatrix} \cos\frac{\pi}{2} & -\sin\frac{\pi}{2} & 0 & 0 \\ \sin\frac{\pi}{2} & \cos\frac{\pi}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

$$\mathbf{C} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Given the point $P = (1, 2, 3, 1)$, $P' = \mathbf{ABC}P = (-7, 3, 10, 1)$.

b) If the matrix $\mathbf{B}$ is used to transform points of an object, what matrix is used to transform the corresponding surface normals? What is the direction of the normal $\mathbf{n} = (1, 0, 0)$ after this transform?

In general, if a matrix $\mathbf{M}$ is used to transform points, then $\mathbf{M}^{-\top}$ is used to transform normals. In our case:

$$\mathbf{M}^{-\top} = \mathbf{B}^{-\top} \tag{4}$$

Also note that $\mathbf{B}^{-\top} = \mathbf{B}$ because $\mathbf{B}$ is a rotation matrix, and rotation matrices are orthogonal, i.e., $\mathbf{R}^{-1} = \mathbf{R}^{\top}$.

Now compute $\mathbf{M}^{-\top}\mathbf{n} = \mathbf{R}_z(90)\mathbf{n} = (0, 1, 0)$. Thus, the direction of the normal after transform is $(0, 1, 0)$.

c) Derive the view matrix for a camera placed at $E = (2, 2, 2)$ looking at the point $C = (2, 0, 2)$ and with an up-vector $(1, 0, 0)$. Solution: See Lecture 5.

$$\text{View} = \begin{bmatrix} 0 & 0 & 1 & -2 \\ 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

d) Describe the steps needed to transform the triangle from A to A' and B to B' in Figure 1,

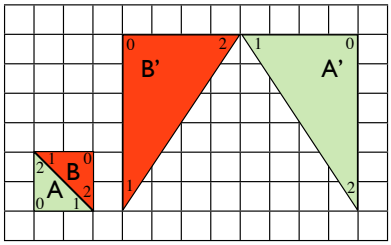

Figure 1: Transform from A to A' and B to B'

One possible solution:
The origin is placed in the lower left corner of triangle A, and assume that Figure 1 shows the xy-plane. First, scale triangle A with two in x and three in y. Then, rotate the triangle 180 degrees around the origin (around the z-axis, i.e., using the $\mathbf{R}_z$ rotation matrix). Finally, translate along the vector (11,6,0). The matrix applied to the triangle at A is then:

$$\mathbf{T}(11, 6, 0)\mathbf{R}_z(180)\mathbf{S}(2, 3, 1) \tag{5}$$

The full matrix is given by:

$$\mathbf{M} = \mathbf{T}(11, 6, 0)\mathbf{R}_z(180)\mathbf{S}(2, 3, 1) = \begin{bmatrix} -2 & 0 & 0 & 11 \\ 0 & -3 & 0 & 6 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6}$$

Similarly, at triangle B, first scale x with three, and y with two, rotate 90 degrees around the z-axis, and translate seven units along the positive x-axis.

$$\mathbf{N} = \mathbf{T}(7,0,0)\mathbf{R}_z(90)\mathbf{S}(3,2,1) = \begin{bmatrix} 0 & -2 & 0 & 7 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

In RenderChimp, each node has a **TRS** transform matrix, where the matrices are applied from right to left. For this example, given that triangle A is defined with the three vertices $P_0 = (0,0,0,1)$, $P_1 = (2,0,0,1)$ and $P_2 = (0,2,0,1)$, we can write:

```
triA->setScale(2,3,1);
triA->setRotateZ(M_PI);
triA->setTranslate(11,6,0);
```

Similarly, for triangle B with vertices $P_0 = (0,2,0,1)$, $P_1 = (2,0,0,1)$ and $P_2 = (2,2,0,1)$ we have

```
triB->setScale(3,2,1);
triB->setRotateZ(M_PI/2);
triB->setTranslate(7,0,0);
```

Note that if we want to multiply the matrices together in *another* order, e.g., **SRT**, or add a second translation, we must create new nodes in RenderChimp to handle this case.

3. Interpolation and Graphics Techniques

a) Smoothstep is cubic interpolation between [0,1] on the interval $x \in [a,b]$, with the constraint that the derivative is zero at $x = a$ and $x = b$. Derive the cubic interpolant.
Set $t = \frac{x-a}{b-a}$, and denote the cubic interpolant $f(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3$. The values and derivatives at $t = 0$ and $t = 1$ give us the four equations:

$$\begin{aligned} f(t=0) = 0 &\Rightarrow c_0 = 0 \\ f(t=1) = 1 &\Rightarrow c_0 + c_1 + c_2 + c_3 = 1 \\ f'(t=0) = 0 &\Rightarrow c_1 = 0 \\ f'(t=1) = 0 &\Rightarrow c_1 + 2c_2 + 3c_3 = 0 \end{aligned}$$

Now use these four equations to solve for the four coefficients c_i. In general, this is a system of equations $\mathbf{Ax} = \mathbf{b}$ that can be solved using a matrix inverse, i.e., $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, but in this case, both c_0 and c_1 are zero, so we are left with the two equations $c_2 + c_3 = 1$ and $2c_2 + 3c_3 = 0$. Solving for c_2 and c_3, we get $c_2 = 3$ and $c_3 = -2$, and the cubic interpolant is then:

$$f(t) = 3t^2 - 2t^3,$$

where $t = \frac{x-a}{b-a}$.

b) Explain the bump mapping algorithm.
Derive tangent space: Compute tangent (**t**), binormal (**b**) and normal (**n**) in vertex shader and pass to pixel shader. These three vectors represents *tangent space* a unique coordinate system for each point of the object.
Find tangent space normal: Lookup coordinate vector (α,β,γ) from a texture, and remap from [0,1] to [-1,1] (Colors in GLSL: [0,1] instead of [0,255]).
Note that this lookup is performed in the pixer shader, to give a perturbed normal for each pixel covered by a triangle. Now, express the perturbed normal in object space (if the tangent (**t**), binormal (**b**) and normal (**n**) are expressed in object space) as:

$$\mathbf{n}' = \alpha\mathbf{t} + \beta\mathbf{b} + \gamma\mathbf{n} \tag{8}$$

Finally, transform the perturbed normal $\mathbf{n}'$ from object space to world space (or the space you've chosen to shade in) and use this normal for shading. See Lecture 4 for details.

c) Reflection mapping is an approximation of true reflections. Why? Motivate your answer.
Reflections are simulated by indexing in a texture based on the direction of the reflected view vector. The result is approximate reflections from distant objects, and reflections in near-by objects, or the object itself are not captured. Furthermore, the reflection map is only correct from *one unique point of the scene*, i.e., the point the cube map was generated from. Commonly, the cube map is static, or at least not updated for each frame. Finally, the spherical reflection map is often represented with a cube-map. where the sphere is approximated with six faces. True specular reflections can be simulated by tracing a ray along the incoming ray's reflection direction and compute wich object this ray first intersects.

d) What is the difference between an orthographic and perspective projection?
See Lecture 5, slide 16 for a visual difference. Perspective projections have so called perspective foreshortening, where you divide the coodinates with the depth. Objects further away become smaller. In othographics projection, you simply drop the depth coordinate. Orthographic: $(x, y, z) \mapsto (x, y)$. Perspective: $(x, y, z) \mapsto (x/z, y/z)$.

4. The Graphics Pipeline

a) Describe the stages in a real-time graphics pipeline; from input to a resulting image. Include the different stages of the pipeline and their responsibilities.

Application setup:
Setup geometry, shaders and transform matrices (done on the CPU)
For each triangle (done on the GPU): Apply transforms (e.g., MVP) in vertex shader Project on screen (divide by w coordinate) Rasterize: Evaluate edge equations at pixel center, to determine which pixels are covered.
For each covered pixel: Depth buffer test to check if closest object If visible: run pixel shader, store in color buffer.
The vertex shader works on vertices and transforms each vertex and its attributes (sent from the application through vertex arrays), such as the vertex normal, texture coordinates, etc. The vertex shader always outputs a position in clip space to the rasterizer (i.e., applies the MVP matrix to each vertex). The transformation matrices are commonly passed as uniforms.
The rasterizer computes visibility, i.e., in which pixels the current triangle is visible using inside tests basedon *edge equations*. It also interpolates the vertex attributes for each pixel. As part of the rasterization phase, the *depth buffer* test is executed to determine which of the triangles overlapping a pixel that is visible. Before rasterization, backface culling may be performed to avoid rasterizing backfacing triangles. Note that a backface test must be done after vertex shading.
The pixel shaders receives the interpolated vertex attributes from the rasterizer and a set of uniforms as inputs and computes a color for the pixel. The pixel shader always outputs a color (unless the `discard` operation is called.

b) What are the most computationally costly parts of the graphics pipeline? Motivate your answer.
This is very scene-dependent. For a scene consisting of one big triangle, the vertex shader work is easy (transform three vertices). The rasterization work is medium, as the triangle covers many pixels (i.e., many inside tests need to be executed), but there is only one triangle, so only depth bufer tests for all pixels covered by one triangle. The pixel shader work is in this case significant, as for all pixels covered by the triangle, a pixel shader program need to be executed. For a scene with many small triangles, the vertex shader work may be signficicant, but the pixel shader work is often still the bottleneck, due to the amount of pixels that needs to be shaded. The rasterizer and depth test are performed in *fixed-function* hardware on a GPU, so although a lot of computations need to be performed there, it can be done efficiently. If the rasterization is done in software, however, the rasterization cost is significant. The cost is also heavily dependent on the complexity of the shader programs. A pixel shader may contain several hundred lines of code, and in practice, the pixel shader work is often the limiting factor in a modern game. Moving computations from pixel to vertex level is a common optimization technique. Gouraud shading is one example.

c) In Figure 2, three primitives are rendered in order: A,B,C. Explain how depth buffering works, and how it can correctly render the three triangles. If the triangles were instead rendered in order C,A,B, with depth test active, what would the image look like?

The image would look the same, regardless of the order the primitives are rendered. See Lecture 6 for a summary of depth buffering.
For each pixel: store a depth value d_{stored} (initialize to large value). For each pixel: compute depth value d_{new}, of current triangle at hitpoint If $d_{\text{new}} < d_{\text{stored}}$ we have a hit. Update the depth buffer: $d_{\text{stored}} := d_{\text{new}}$, and call the pixel shader. Otherwise, the triangle is covered by already drawn primitives and we are done and can move to next pixel.

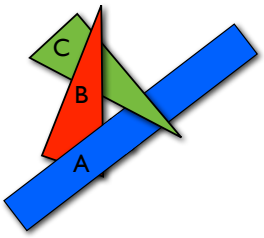

Figure 2: Three overlapping primitives.

5. The rendering equation

a) Describe and discuss each term of this equation. Also describe why the equation is so hard to solve.
See Lecture 6 for definitions of the included terms. The reason the equation is so hard to solve is because it contains the light intensity I on both the left and the right side, and must be recursively evaluated *for all points in the scene* for a correct solution. This is not tractable, and in practice, this equation is approximated or evaluated at a subset of points. Furthermore, determining mutual visibility between two points in the scene is a complex operation if the scene contains many objects. Also, the BRDF may be complex and hard to evaluate, and may vary for each point in the scene.

b) Describe how the rendering equation can be simplified in order to arrive at the Phong shading model
See Lecture 6. Simplifications: Disregard from the secondary bounces of light. Unfold the recursion on the rendering equation once, and only include emission from the second step. In other words, only accumulate light *directly from light sources*, not from all points in the scene. Assume perfect visibility and non-emitting surfaces. As BRDF, use the simple Phong model.

c) Mention at least two effects that are hard to reproduce in real-time graphics, but needed for high quality offline rendering. Motivate your answer.
Realistic reflections and refractions, indirect lighting (lighting from secondary bounces) etc. These effect are more easily expressed in a ray tracing engine, that more carefully approximates the rendering equation.
Hair, fur and fine scale geometry. These effects often require millions of triangles, and does not often fit in the budget of a real-time graphics application (with a frame time of 20 ms).
Volumetric effects, such as realistic fog, clouds and fire is often more coarsely approximated in real-time graphics due to limited resources.

6. Hierarchical Modeling, Toon Shading and Shadows

a) Describe a scene graph for the model and use the scene graph to animate the vertical ascent (take-off) of the helicopter.
The scene graph could look like in Figure 3.

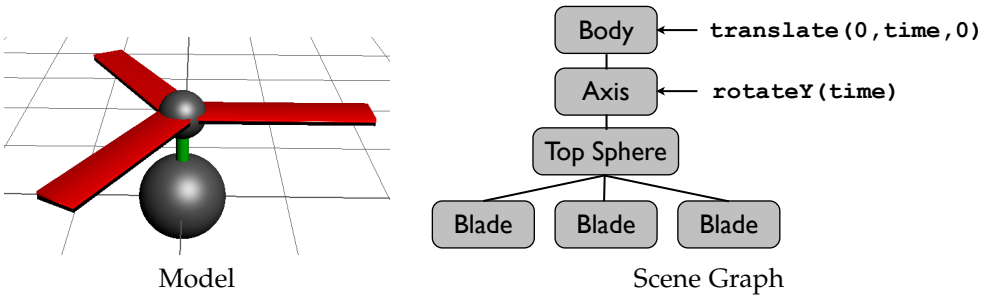

Figure 3: A simple helicopter model and corresponding scene graph

b) Design a toon shader: The silhouette can be detected by looking for when $|\mathbf{v} \cdot \mathbf{n}|$ is small (the view vector nearly perpendicular to the surface normal). Replace the diffuse shading with a constant color, and use a brighter constant color near the specular peak.

```
uniform float shininess;
in vec3 fN;  // Normal
in vec3 fL;  // Light vector
in vec3 fV;  // View vector
out vec4 fColor;

void main()
{
    vec3 N = normalize(fN);
    vec3 L = normalize(fL);
    vec3 V = normalize(fV);
    vec3 R = normalize(reflect(-L,N));

    vec3 color           = vec3(0.0, 0.0, 1.0); // blue
    vec3 specularColor   = vec3(0.5, 0.5, 1.0); // light blue
    vec3 silhouetteColor = vec3(0.0, 0.0, 0.0); // black

    if (pow(max(dot(V,R),0.0), shininess) > 0.5)
        color = specularColor;

    if (abs(dot(V,N)) < 0.3)
        color = silhouetteColor;

    fColor = vec4(color, 1.0);
}
```

c) Discuss how you could simulate shadows, both in a ray tracer and in a rasterization pipeline. Focus mostly on the latter.

In a ray tracer, shadows could be simulated by tracing rays from the point towards the light source. If there is an object on the ray between the point and the light source, the point cannot receive light from that light source, and the point is in shadow from that light source.

In a rasterizer, one way of simulating shadows is the so-called *shadow mapping* technique.
First, place the camera at the light source, looking along the light source direction (trivial if it is a directional or spot light, for point lights it is more involved). Now, render an image from the point of the light source, using a MVP_{light} matrix to transform geometry into the space of the light source, and for each pixel, store the depth of the closest object (this is what is also stored in the depth buffer when all geometry has been rendered).

In a second pass, render from the "real" camera. Now bind the depth buffer obtained from the previous pass as a texture (black & white image that represent the depth at each pixel from a rendering from the light's point of view). This texture is often called the *shadow map*.

Now, when shading a pixel, we want to use this depth texture to perform a test: "If the current point that we shade is further from the light than the value of the shadowmap at the same point, this means that the scene contains an object that is closer to the light." In other words, the current pixel is in shadow.

The tricky part is to express the current point, P, we shade, in the space of the shadow map (i.e., a frame where the light source is at the origin), otherwise the test above does not make sense. To express P in the frame of the light source, we apply the MVP_{light} matrix that we used from the first rendering pass to transform the point into the coordinate frame used for the shadow map.

$$P_{\text{shadow}} = MVP_{\text{light}}P. \tag{9}$$

This transform can be performed in the vertex shader, and P_{shadow} is passed along to the pixel shader.

After this matrix has been applied, we can perform the shadow test in the pixel shader.

```
float visibility = 1.0;
if ( texture( shadowMap, Pshadow.xy ).z  <  Pshadow.z){
    visibility = 0.0;
}
```

Now we can use the visibility value to modify our shading value, for example, to multiply the diffuse and specular components with.

Remarks: In practice, $P_{\text{shadow}} = MVP_{\text{light}}P$ is expressed in NDC, which goes from $[-1, 1]$ in x and y, but a texture in GLSL is indexed between $[0, 1]$, so before performing the texture lookup, we must shift and bias the xy coordinates $[-1, 1] \mapsto [0, 1]$, e.g., $x' = x/2 + 1$ and $y' = x/2 + 1$. This final step was omitted for clarity above.
Note that we also need to perform $P_{\text{ndc}} = MVP_{\text{camera}}P$ as usual, and output that position from the vertex shader.

The end.