

Pool Billiard

An OpenGL-based billiard simulation

Stefan HUBER
Kamran SAFDAR
Andreas SCHRÖCKER

Fachbereich Computerwissenschaften
Universität Salzburg

June 10, 2009

Table of Contents

- 1 Modelling the Balls
- 2 Modelling the Table
- 3 Force Simulation
- 4 Viewing
- 5 Animation
- 6 Control

- 1 **Modelling the Balls**
 - An Introduction to Texture Mapping
 - What is Texture Mapping?
 - 2D Texture Mapping
 - Texture Mapping a Sphere
 - Texturing the Billiard Balls
 - Drawing the Billiard Balls using OpenGL
- 2 Modelling the Table
- 3 Force Simulation
- 4 Viewing
- 5 Animation

The Ball

- Rendered using a sphere (`gluSphere`) with unit radius, 50 slices and 50 stacks.
- A texture image is wrapped around it to give it a look of billiard ball.

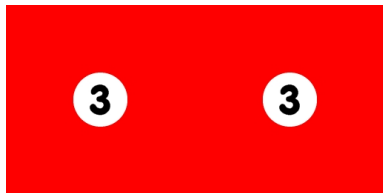


Figure: Texture image

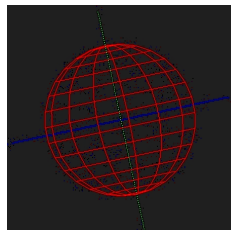


Figure: Wire Sphere

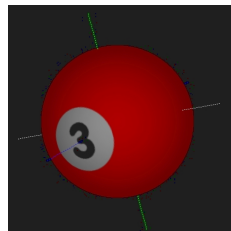


Figure: Textured Sphere

Texture Mapping

- Mapping of a function, called a *texture map*, onto a surface.
- Can be understood as gluing a 2D image onto the surface of a 3D object.
- Pioneered by Edwin Catmull in 1974.
- Can be used to enhance the aesthetic appearance of an object at a relatively small computational cost.

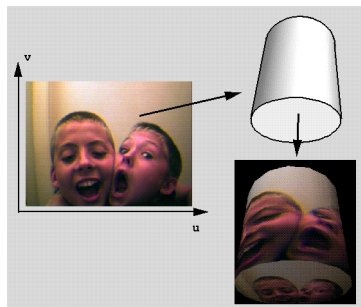


Figure: A 2D image mapped onto a 3D cylinder.

Image source: <http://prosjekt.ffi.no/unik-4660/lectures04/chapters/Basic.html>

- The 2D texture space is defined in terms of parameters s and t each ranging from 0 to 1.
- For each point in the 3D object space, a corresponding point in the 2D texture space, called a *texel*, is mapped.
- While rendering the 3D object, each point (x, y, z) is assigned a color value computed from the corresponding 2D texel value (s, t) .

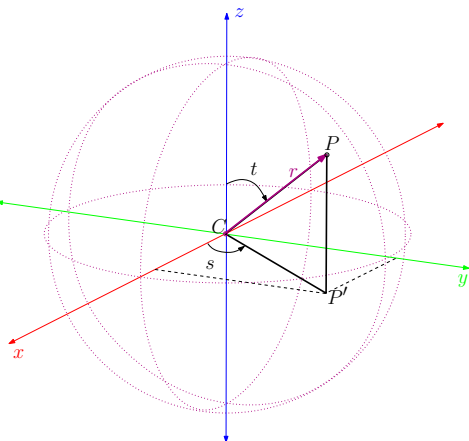
Geometry of a Sphere

- Geometrically stating, a sphere with radius r centred at $C(x_c, y_c, z_c)$ is the locus of points

$P_i(x, y, z), (0 \leq i \leq n)$, such that

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$$

- For any point $P(x, y, z)$ on surface of the sphere, let s be the angle from $+x$ -axis to $\overrightarrow{CP'}$, and t be the angle from $+z$ -axis to \overrightarrow{CP} , such that $(0 \leq s \leq 2\pi)$ and $(0 \leq t \leq \pi)$.



Geometry of a Sphere

- The parametric representation can be explained mathematically as:

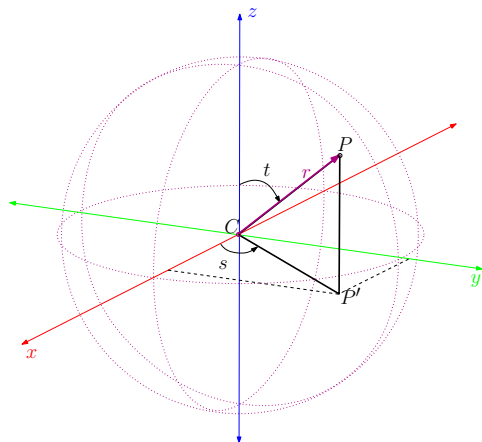
$$x = x_c + r \cdot \sin t \cdot \cos s$$

$$y = y_c + r \cdot \sin t \cdot \sin s$$

$$z = z_c + r \cdot \cos t$$

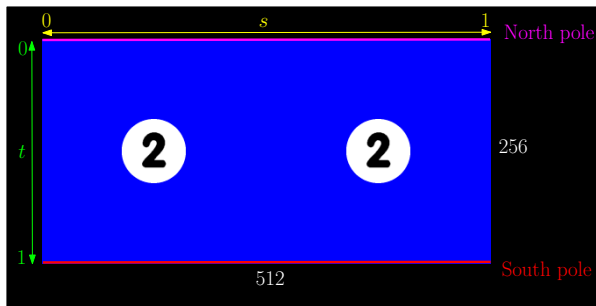
with $s \in [0, 2\pi]$ and $t \in [0, \pi]$

- So, if we know the coordinates of a point $P(x, y, z)$ on surface of the sphere, then we can easily find out the values of parameters s and t .



Billiard Ball Textures

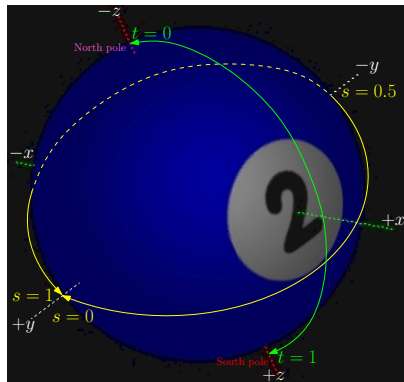
- The textures are 512×256 RAW images.
- The lines marked as **North pole** and **South pole** are mapped onto single points at the poles.



Mapping the Ball Textures

- Texture coordinates are assigned such that:

$s = 0.00$	at	$y = +r$		$t = 0.0$	at	$z = -r$
$s = 0.25$	at	$x = +r$		$t = 1.0$	at	$z = +r$
$s = 0.50$	at	$y = -r$				
$s = 0.75$	at	$x = -r$				
$s = 1.00$	at	$y = +r$				



Draw the Billiard Ball

- Load the texture map in memory
- Create a quadric object
- Enable texturing of the quadric object
- Draw a quadric Sphere, generate texture coordinates for it as per the scheme already known and load the texels
- Do garbage collection and house keeping

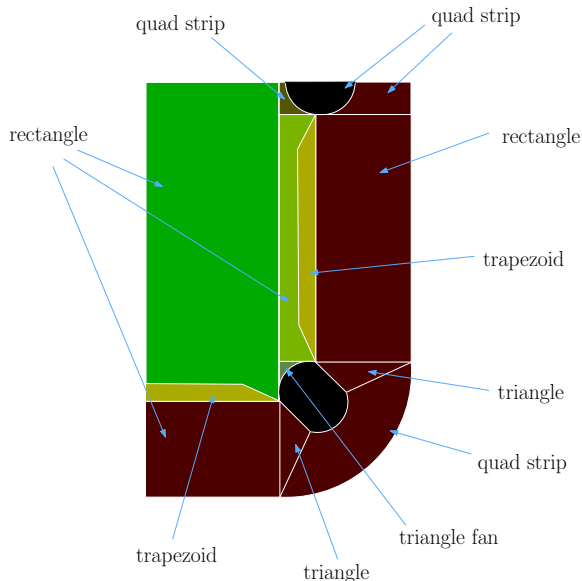
Drawing the Billiard Balls using OpenGL

```
1  void DrawBall()
2  {
3      GLuint texture_map;
4      GLUquadric *qobj = gluNewQuadric();
5
6      texture_map = LoadTextureRAW("textures/2.raw", true, 512, 256);
7      glEnable(GL_TEXTURE_2D);
8      glBindTexture(GL_TEXTURE_2D, texture_map);
9      gluQuadricTexture(qobj, GL_TRUE);
10     gluSphere(qobj, 1, 50, 50);
11     gluDeleteQuadric(qobj);
12     glDeleteTextures( 1, &texture_map );
13     glDisable(GL_TEXTURE_2D);
14 }
```

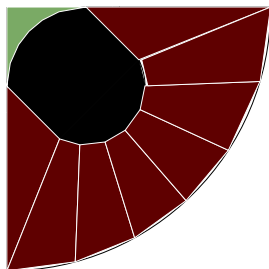
- 1 Modelling the Balls
- 2 Modelling the Table
 - Drawing the Table
 - Texturing the Table
- 3 Force Simulation
- 4 Viewing
- 5 Animation
- 6 Control

Structure of the Table

- The Table is axis aligned.
- Rectangular planes
 - GL_QUADS
- Pockets
 - GL_QUAD_STRIP
 - GL_TRIANGLE_FAN
 - GL_TRIANGLES



- Draw a half of and a quarter of a circle with brute-force scan-conversion algorithm
- Save the points in two arrays
- Connect the points with `GL_QUAD_STRIP`
- For the hole: Draw an arc and use `GL_TRIANGLE_FAN`



- A cylinder is drawn
`void gluCylinder()`
- And closed with
`void gluDisk()`



Symmetric Construction of the Table



- We make use of the symmetry in the table's geometry.
- One quarter of a table is drawn 4 times with `glScalef()`.



- Two different raw files: Wooden texture and green cloth texture
- The texture coordinates are specified with `glTexCoord2d(u, v)`
- For `GL_QUADS`: Set `u,v` to `(0.0,0.0)`, `(0.0,1.0)`, `(1.0,0.0)`, `(1.0,1.0)`
- For the rest: Calculate `u,v` so that it looks properly

- 1 Modelling the Balls
- 2 Modelling the Table
- 3 Force Simulation**
- 4 Viewing
- 5 Animation
- 6 Control

- Orthogonal projection for drawing in 2D

```
glOrtho(0, 10.0, 0.0, 10.0, -1, 1);
```

- We used three `GL_QUADS` with different blending functions

- Black background:

```
glBlendFunc (GL_ZERO, GL_ZERO)
```

- Color slide: `glBlendFunc (GL_ONE, GL_ZERO)`

- Transparent slide:

```
glBlendFunc (GL_DST_COLOR,  
GL_SRC_ALPHA)
```



- 1 Modelling the Balls
- 2 Modelling the Table
- 3 Force Simulation
- 4 Viewing
 - Camera Movement
- 5 Animation
- 6 Control

A Top View



A Strike Direction View



Looking at the Center of the Table



Setting Camera in Display Function

```
1 switch (nCameraState)
2 {
3     case CAM_SHOOTING:
4         gluLookAt(eye_x, eye_y, eye_z,
5                 arrBalls[0].pos.x,0.285,arrBalls[0].pos.z,
6                 0.0, 1.0, 0.0);
7         break;
8
9     case CAM_OVERVIEW:
10        gluLookAt(0,35,0, 0,0,0 , 1.0, 0.0, 0.0);
11        break;
12
13    case CAM_CENTER:
14        gluLookAt(eye_x, eye_y, eye_z, 0,0,0, 0.0, 1.0, 0.0);
15        break;
16 }
```

- 1 Modelling the Balls
- 2 Modelling the Table
- 3 Force Simulation
- 4 Viewing
- 5 Animation**
 - Movement and Friction
 - Rotation of Balls
 - Collision with the Cushion
 - Collision of Balls
- 6 Control

- Movement: Adding a constant vector to the position.
- Friction: Simulated by subtracting a part of the movement vector.

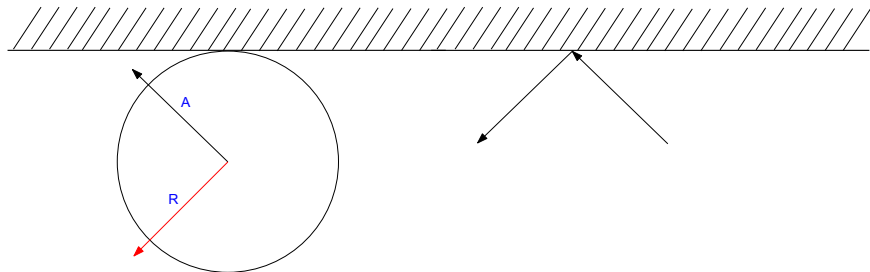
- In real life, movement of a ball is understood as:
 - Displacement from start position
 - Revolution around its center in the direction of movement
- Rotation matrix is stored for every ball and applied when drawing - on every step of animation a new rotation matrix is multiplied on this stored matrix

$$\begin{pmatrix} x^2(1-c) + c & xy(1-c) - zs & xz(1-c) + ys & 0 \\ yx(1-c) + zs & y^2(1-c) + c & yz(1-c) - xs & 0 \\ xz(1-c) - ys & yz(1-c) + xs & z^2(1-c) + c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

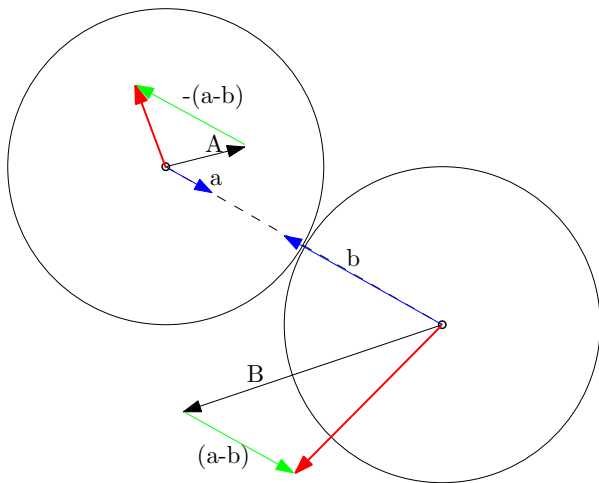
$$c = \cos(\text{angle}), s = \sin(\text{angle}), \|(x, y, z)\| = 1$$

Collision with the Cushion

- The ball is reflected back when it strikes the cushion.
- Reflection is calculated by changing the sign of x or z component of the movement vector (remember, cushions are parallel to x or z axis).



Collision of Balls



Collision of Balls

```
1 void ball_check_collision(struct BALL* b1, struct BALL* b2) {
2     GLdouble b1b2_norm;
3     Vertex v1, v2, vdiff, b1b2;
4
5     vec_diff(&b2->pos, &b1->pos, &b1b2);
6     b1b2_norm = norm(&b1b2);
7     if (b1b2_norm < BALL_DIAMETER) {
8         GLdouble s1 = vec_dotp(&b1->move, &b1b2)/b1b2_norm;
9         GLdouble s2 = vec_dotp(&b2->move, &b1b2)/b1b2_norm;
10        GLdouble sd = fabs(s1-s2);
11
12        vec_mult(&b1b2, 1/b1b2_norm, &b1b2); // build unit-vector of b1b2
13
14        vec_mult(&b1b2, -sd, &v1);
15        vec_mult(&b1b2, sd, &v2);
16
17        vec_add(&b1->move, &v1, &b1->move); // new speed of the balls
18        vec_add(&b2->move, &v2, &b2->move);
19    }
20 }
```

- 1 Modelling the Balls
- 2 Modelling the Table
- 3 Force Simulation
- 4 Viewing
- 5 Animation
- 6 Control**

- Left and right arrow keys: Rotate around the y-axis
- Up and down arrow keys: Move towards the ball and backwards (eye_y does not change)
- The keys 'y' and 'Y': Decrease and increase eye_y
- Left mouse button: Push the ball