You may choose one of two project titles, "Air Hockey Game" and "GPU-based Ray Tracing", recommended for your project. Some necessary information and requirements for these two projects are provided as follows:

1. Air Hockey Game

I. Introduction

This project is based on an arcade game called "Air Hockey" found in JumPin Gym U.S.A ("美国冒险乐园" in Chinese). This is a 2-player game playing on a table with goals on opposite sides (see Fig.1). Each player will use a piece of "Mallet" to push a disc (called Puck) into other's goal and at the same time protect one's goal from the Puck. Apart from the two goals, the table is surrounded by walls which bounce the Puck and prevent the Puck goes outside.

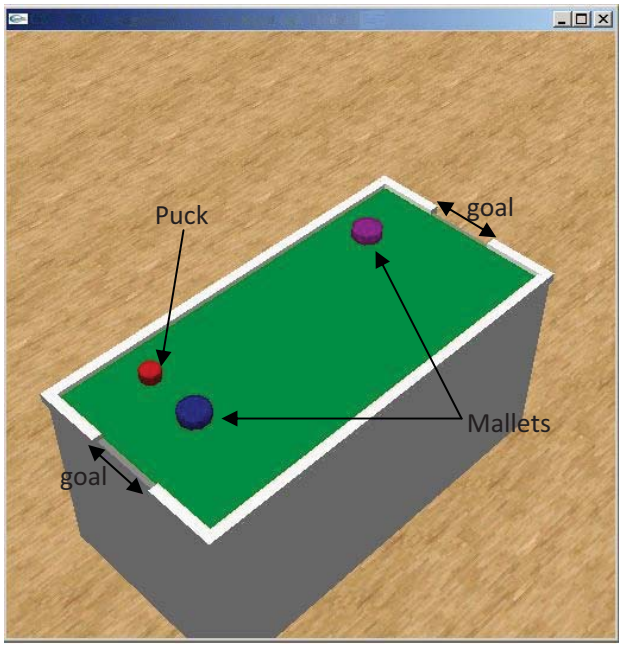


Fig. 1 The screenshot of the demo program

In this project, you are going to implement this arcade game on a PC. The program will just

support single player, so the opponent will be controlled by computer. The same as the previous assignments, you are required to complete this project using OpenGL for the graphics display part. While you are free to use any data structures in your program.

II. Implementation Details

In the project package, there is a demo program (i.e., *demo.exe*) and a file (i.e., *readme.txt*) indicating the keyboard usage of the demo.

All programs should meet the reasonable programming standards: header comments, in-line comments, good modularity, clear printout, efficiency.

To complete this project, we can divide it into 4 tasks as follows:

Task 1: Draw the 3D objects – Game Table, Puck, Mallets and Textured Floor. (20 points)

- Draw 5 3D objects : Table, floor, Puck and 2 Mallets

Table : The table has 3 different parts : Playing area, Wall and Goal.

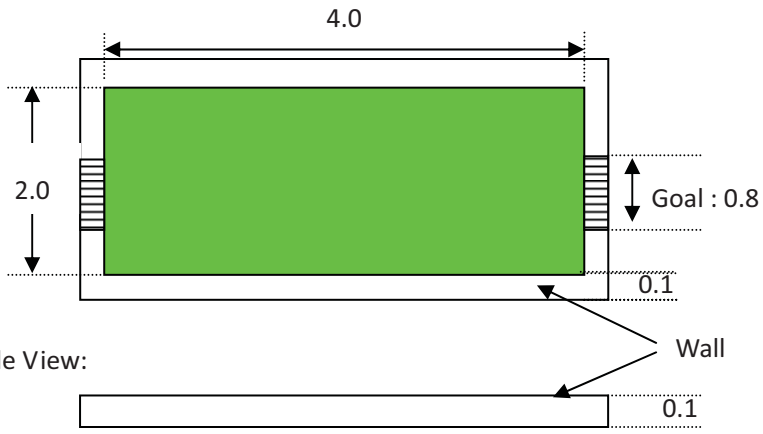
Playing area (in green below) is a rectangle where the Puck can move, it should be 2:1 in dimension.

Wall surrounds the Playing area except at the 2 Goals. It should be about 0.1 thick and 0.1 in height vertically (see side view below).

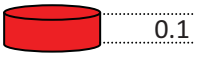
2 Goals are at the center of 2 shorter sides with around 0.8 in length.

Reference design of the Table (which is used in the demo),

Top View:



Floor: A large texture-mapped rectangle (choose any texture or floor.bmp in demo program).



Puck: A short cylinder with diameter about 0.2 and height about 0.1.

Mallet: A short cylinder with diameter about 0.3 and height about 0.1. Two Mallets are in different colors.

Task 2: Dynamics of Puck. (30 points)

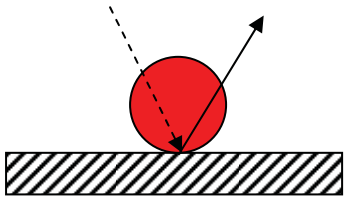
The Puck should move and bounce back when it hits the walls or the players' Mallets. When the game starts, the Puck should move in the direction towards either player's goal with some random deviations.

Note that the movement of Puck should be updated in system time interval (e.g. 10 ms) instead of arbitrary time interval (e.g. as fast as possible), since computers may vary in the update rate if not using the system time.

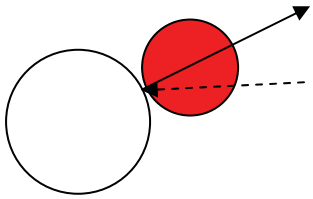
Collision detection should be preformed every time interval. If a collision is detected, the Puck will **bounce back** and its moving direction will change. Since all these actions occur on the table which is a 2D plane, you can reduce all calculations from 3D to 2D.

The dynamics of Puck should obey the physical laws. You may assume that neither air resistance and energy loss nor external force will be applied on the Puck. As a result, the bouncing will be as illustrated below:

Collision with Wall: Always bounce like reflection

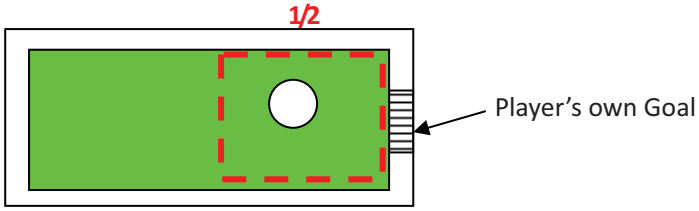


Collision with Mallet: Always bounce back in the direction parallel to the normal of the Mallet at contact point.



Task3: Player's control and simple AI of opponent. (30 points)

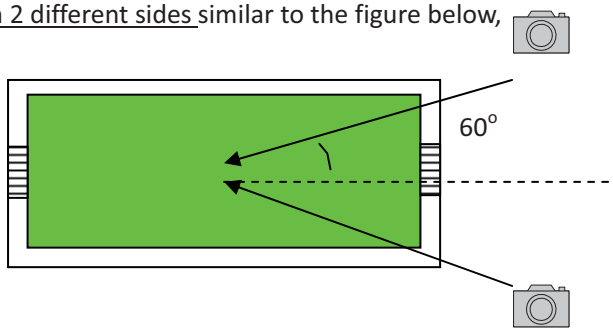
You should enable a human player to control the Mallet by mouse. Player's Mallet should move to the position where the mouse cursor is pointing. However, the movement of Mallet should be restricted in the player's own half of the table. As illustrated below:



You have to implement simple AI for the computer opponent. It is not required that the opponent is as wise as a human player, but it should have some response to block the Puck. It is always easy to have a "perfect" computer opponent who can block every coming Puck, so you should add some random factors to **allow computer losing.**

Task4: Change of view, Game flow and your feature (20 points)

The program should allow the user to change view angle to the table by pressing **LEFT and RIGHT arrow** buttons. You can limit the view angle (say 60^o), but at least you should allow user to view the table from 2 different sides similar to the figure below,



When the Puck goes into the goal, the game should check whether the player has **WON or LOST** and shows the corresponding sentence like "You Win" or "You Lose" on the center of the screen. Then, the game play should be stopped. The game can be restarted by pressing **ENTER button.**

There are 5 marks reserved for an addition feature. You are free to implement any feature, but you must mention it in the header of your source code (game.c) or a separate readme file. Here are some suggestions for feature:

1. Enhance the objects' appearance by using texture mapping or other techniques.
2. A score board showing number of win and lose.
3. A more aggressive computer opponent (but should not be a prefect player)

lll. Grading Scheme

Your project will be graded by the following marking scheme:

● Draw Objects – table, 2 mallets, puck and floor	20%
● Collision and Bounce between puck and walls	15%
● Collision and Bounce between puck and Mallets	15%
● Control player's Mallet by mouse	20%
● Simple AI of opponent	10%
● Change of view and Game flow	15%
● Creativity (Additional Feature)	5%
● **Total**	**100%**

Note: No marks will be given if the program displays nothing; even you had other parts done.

IV. Additional guidelines to submit your programs

1) You are suggested to write your programs on Windows, since there are enough technical supports available from us or the Internet. We have also provided a *C/C++ rendering programming environment* in the compressed file, "*A C&C++ rendering programming environment.rar*", for your final project. Its user guide can be found in Appendix 2.

   If you develop the program in other platforms, make sure your program can be compiled and executed on Windows as the program will only be tested on this platform.

2) Zip the source code file (e.g., *game.c*), the readme file (e.g., *readme.txt*) and any additional bmp files in a .zip or .rar file. Name it with your own name (e.g. ZhangSan.zip). That is, there should be at least **two** files in your submitted package.

3) ***Fail the course if you copy the project result from others.***

2. GPU-based Ray Tracing

I. Introduction

It is necessary that this project be completed by using a GPU-accelerated ray tracing scheme. The programs must be written with OpenGL + Cg/CUDA. The 3D geometric models to be rendered should be generated and inputted from a data file, which may be either the output of a certain CAD software system like 3D Studio Max, AutoCAD or Maya etc., or edited by yourself.

II. Implementation Details

For your final project, we provide a C/C++ rendering programming environment, whose

project is built on Visual Studio 2008 with Cg, OpenGL and GLEW. Before opening the project, you should first install OpenGL V2.0 or later version that can be downloaded from NVIDIA'S Website and GLEW.lib V1.5 that is compressed in the RAR file. If you still don't know how to do this, see Appendix 1.

Below is an example rendered by us —— the scene of the 3D Sponza model with a Venus statue in the middle and a plane mirror in front of the statue. (See Fig.1)

Fig.1 Sponza with Venus

Right now, if you compile and link the project, and then run the demo with the command " OpenGLSceneLoader.exe sponzaModel.txt", you won't see the mirror's reflection in the image, but you will see a white rectangle in the mirror area. The source code of this project includes several parts. More information about the environment can be found in Appendix2. In fact what you should really concern is to reload the function "void draw()" in the Scene class definition to implement ray tracing. We suggest that you use Cg or GLSL to achieve a GPU-based ray tracer.

lll. Grading Scheme

Your project will be graded by the following marking scheme:

- Draw Objects –furniture, mirror, transparent object, walls, floor and ceiling 20%
- texture-mapping 5%
- GPU-based ray tracer 50%
- Interactively changing the position and direction of the view point 15%
- Creativity (Additional Feature) 10%

- **Total** **100%**

Note: No marks will be given if the program displays nothing; even you had other parts done.

IV. Additional guidelines to submit your programs

1) You are suggested to write your programs on Windows XP/7, since there will be enough technical support. If you developed the program in other platforms, make sure your program can be compiled and executed on Windows XP/7 as the program will only be tested on this platform.

2) Zip the source code file (e.g.,scene.cpp), the readme file (e.g., *readme.txt*) and any additional bmp files or others in a .zip or .rar file. Name it with your own name (e.g. ZhangSan.zip). That is, there should be at least **two** files in your submitted package. Note: you should tell us where is your implementation in the readme file.(e.g., line 75-299 in scene.cpp, function raytracing())

Fail the course if you copy the project result from others.

Appendix 1. Set up the OpenGL Programming Environment on Visual Studio 2008

A1.1 Install OpenGL & GLUT (Omitted)

A1.2 Install GLEW

1) Unzip the file glew1.5.7.rar;
2) Put the file "glew32.dll" into the same directory as your executable file, or "C:\WINDOWS\system32"
    - Put the file "glew.h" into the standard Visual C++ include directory (e.g., "C:\Program Files\Microsoft SDKs\Windows\v7.0A\include\gl" for the Visual Studio 2010, or "C:\Program Files\Microsoft SDKs\Windows\v6.0A\Include\gl" for Visual Studio 2008)
    - Put the file "glew32.lib" into the standard Visual C++ library directory (e.g., "C:\Program Files\Microsoft SDKs\Windows\v7.0A\Lib" for Visual Studio 2010, or "C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib" for Visual Studio 2008, where there should be lots of lib files such as "opengl32.lib" and "glu32.lib".

3) Make sure your Visual C++ project links in the GLEW library directory. This is located in:
    - Menu: "Project -> (your-project-name) Properties"
    - Tab: "Configuration Properties -> Linker -> Input"
    - Under "Additional Dependencies", add "glew32.lib "
4) #include < GL/glew.h > in your program.
    - Note: This needs to come *after* you #include < stdio.h > and < stdlib.h >.
    - Note: This needs to come *before* you #include < GL/glut.h >.
5) You should *not* include windows.h or any other Windows-specific header files, no matter what you may read on random forum postings!
6) If you get compilation errors because of multiple conflicting definitions of "exit()", then "stdio.h" and "glew.h" have been #include'd in the wrong order. You may fix this by:
    - Reordering your #include files (see step #7). This is the "right" way.
    - Add "#define GLUT_DISABLE_ATEXIT_HACK" to glew.h on the line immediately after the first "#if defined(_WIN32)".
7) If you happen to have a 64-bit version of Windows and Visual Studio, make sure you compile a 32-bit executable.

A1.3 Install Cg

Note: If you use only GLSL shader language, you need not install CG.

8) Download Cg or use “Cg-2.2_February2010_Setup.exe”.
9) Double click the ““Cg-2.2_February2010_Setup.exe” and set up the following instruction.
10) Make sure your Visual C++ project will include the Cg directory. This is located in:
    - Menu: "Project -> (your-project-name) Properties"
    - Tab: "Configuration Properties -> c/c++ -> Reguler
    - Under ”Additional Directory", add “$(CG_INC_PATH)”
11) Make sure your Visual C++ project links in the CG. This is located in:
    - Menu: "Project -> (your-project-name) Properties"
    - Tab:"Configuration Properties -> Linker -> Reguler->Additional Directory"
    - Under ”Additional Directory", add “$(CG_LIB_PATH)”
    - Tab: "Configuration Properties -> Linker -> Input"
    - Under "Additional Dependencies", add "cg.lib cgGL.lib"
12) If you happen to have a 64-bit version of Windows and Visual Studio, make sure you compile a 32-bit executable.

Appendix 2. User guide of the C/C++ rendering programming environment

The input of the program is a scene file. It defines the camera position, lights, materials, as well as 3D models in the scene. All your scene files should be located in the subdirectory “bin\scenes”.

The format of the scene file is a little bit complex. Comment lines are prefaced with a # symbol. First you should define camera properties using key words ”camera pinhole” like:

```
# Setup a camera to view the scene
camera pinhole
        eye 2.78 2.73 -8.00
        up 0 1 0
        at 2.78 2.73 2.795
        fovy 38.5
        res 512 512
 trackb        all
end
```

Then you should define the lights properties using key words” light point” like:

```
light point

        pos    2.78     5.487 2.795
        color  1    0.85 0.43
 trackb        all 0
end
```

You can also define more than one light in your scene using the same description manner. Next, you should define materials using key word “material”. We provide basic Lambertian materials and Phong materals which are implemented in the shader.The following is the two examples.

```
material lambertian green
        albedo 0.15 0.48 0.09
end
material shader red
 vert         phongObjectShader.vert.glsl
 frag         phongObjectShader.frag.glsl
        bind amb      const 0.3 0.1 0.1 1.0
        bind dif      const 0.8 0.05 0.05 1.0
        bind spec     const 0.0  0  .0  0.0   1.0
        bind shiny const 0.0  0.0  0    .0 0.0
 allowsShadows
end
```

You should also define 3D geometric models in the scene. The key word is “object”.We provide several primitives of 3D models.They are trianges, quad, sphere, cylider and mesh.

a) Define triangle:

```
        object tri nameGiveByYou
  v0                ……
```

v1 ……
v2 ……
end

b) Define quads:
object parallelogram nameGiveByYou
v0 ……
v1 ……
v2 ……
v3 ……
end

c) Define sphere:
object sphere nameGiveByYou
cent er ……
radius ……
sta cks ……
s lices ……
end

d) Define cylinder:
Object cyl nameGiveByYou
cent er ……
radius ……
height ……
axis ……
stacks ……
s lices ……
end

e) Define mesh:
object mesh hem nameGivenByYou
fi le fileName
end
Or:
object mesh obj nameGivenByYou
fi le fileName
end

If you are unfamiliar with the ".hem" and ".obj" formats of 3D model files, you can download the format explanations of them via the Internet.

To run the program, you should use the DOS command line "OpenGLSceneLoader.exe sceneFileName.txt" in the directory of the executable file. Note that you need not type the path

of the scene file. When the program runs, it will first read the scene file and load the data . All the data are stored in the scene class variable. The most important member of the scene class is the "geometry" that defines a model list. Every model has a flag which can be used to distinct whether it is reflective.

If the scene is loaded successfully, the program will create a window and show the rendered images in the window. We also predefine a few of interactive operations as follows:

- To change the view point, you can use the left mouse.
- To change the light direction, you can use the right mouse.
- To change the position of the object, you can use the middle mouse.
- To quit the program, you can type "q".

The OpenGL display callback function is defined by *void DisplayCallback(void)* in the file "sceneLoader.cpp". It is the main display function. The project uses the "render to texture" techniques and thus you need to draw your image into the mainWin framebuffer object. See the following code:

```
data->fbo->mainWin->BindBuffer();
data->fbo->mainWin->ClearBuffers();
    glLoadIdentity();//clear the model view
    scene->LookAtMatrix();//set the model view with view point
    scene->SetupEnabledLightsWithCurrentModelview();//enable light
    scene->Draw();// Draw the scene
    //TODO:To implement reflection, you need reload the Draw fuction, or construct a new
function to draw the scene.
data->fbo->mainWin->UnbindBuffer();
```

Currently, we simply call "scene->Draw()" to draw the scene with the Phong reflection model. If you follow the installing step and compile the project successfully, you can run it and see the results.(See Fig.2)

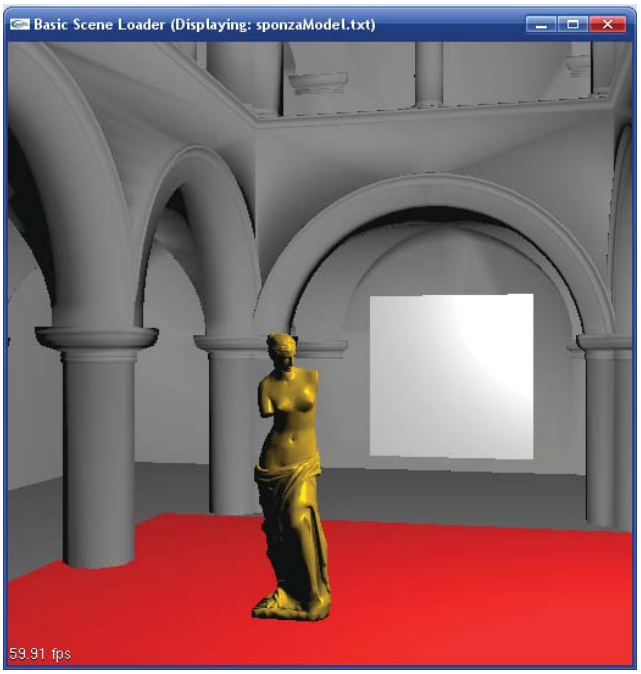


Fig.2

When you rewrite the draw function to implement your ray tracing algorithm, we suggest that you use Cg or GLSL. If you use Cg, you need to follow the following step.

1) In the main function:

a) Init Cg contex

```
shaderContext = cgCreateContext();
```

b) Check the version

```
if (cgGLIsProfileSupported(CG_PROFILE_VP30))
     vertexProfile = CG_PROFILE_VP30;
else{
     printf("Vertex programming extensions (GL_ARB_vertex_program or "
     "GL_NV_vertex_program) not supported, exiting...\n");
     exit(0);
}
if (cgGLIsProfileSupported(CG_PROFILE_FP30))
{
     entry = "basic";
     fragmentProfile = CG_PROFILE_FP30;
}
else{
     printf("Fragment programming extensions (GL_NV_register_combiners2 or "
      "GL_ARB_fragment_program) not supported, exiting...\n");
     exit(0);
}
```

c) Init Cg program and parameters

```
vertexProgram = cgCreateProgramFromFile(shaderContext,
                                        CG_SOURCE, "vertex shader Name",
                                        vertexProfile, "main", NULL);
```

```
cgGLLoadProgram(vertexProgram);
param1 = cgGetNamedParameter(vertexProgram, "param1");
fragmentProgram = cgCreateProgramFromFile(shaderContext,
                                          CG_SOURCE, "fragment shader Name",
                                          fragmentProfile, "main", NULL);
cgGLLoadProgram(fragmentProgram);
param2 = cgGetNamedParameter(fragmentProgram, "param2");
```

2) In the display call back function:

a) Enable the Cg shader

```
cgGLEnableProfile(vertexProfile);
cgGLBindProgram(vertexProgram);
cgGLEnableProfile(fragmentProfile);
cgGLBindProgram(fragmentProgram);
```

b) Your own Draw

… …

c) Disable the Cg shader

```
cgGLDisableProfile(vertexProfile);
cgGLDisableProfile(fragmentProfile);
```

If you use GLSL, you need

1) In the main function:

2) Add a pointer in your structure to the shader in the headfile "renderingData.h"

For example:

```
Struct ShaderData
{
 ……
 GLSLProgram * myShader;
 ……
}
```

a) Init the shader in the InitializeRenderingData()

For example:

```
scene->AddShader( data->shader-> myShader =
                new GLSLProgram( "myShader.vert.glsl",//vertex shader
                                 NULL,//geometry shader
                                "myShader.frag.glsl",//fragment shader
                                true, scene->paths->GetShaderPathList() ) );
```

b) Call the function `InitializeRenderingData();`

3) In the display call back function:

a) Call the function `shader->enableShader()` to enable GLSL shader;

b) Your own drawing code added in *the Draw fuction*

……

c) To disable GLSL shader, call `shader->disableShader();`