RayCore® 1000
Programming Guide

Version 1.0

Contents

Chapter 1 Introduction

3D graphics is one of the computer technologies to generate 3D images. Despite its various benefits in computer graphics, there are several technical barriers to processing 3D images in real applications. One of the examples may be the significant amount of computational costs. Even though this technical challenge has been partly alleviated by the increased performance of computer hardware, 3D graphics users have faced another challenge to enhance photorealistic qualities of images. Among various ideas to overcome the difficulty, ray tracing has been introduced.

Although ray tracing algorithm was first introduced back in 1970s, it has been hardly applied to real applications on real-time basis due to the complexity of the algorithm as well as memory structure required for considerable amounts of computation. In addition to increasing the degree of integration in computer devices, many technical efforts for a high level of parallelization in rendering process should have been implemented to make ray tracing a more practical technology. To realize real-time ray tracing rendering, Siliconarts, Inc. has developed RayCore® 1000, a real-time ray tracing rendering hardware.

1.1 Ray Tracing Algorithm

In computer graphics, ray tracing is a technique that generates an image by tracing the path of lights through pixels in an image plane and by simulating various optical effects such as reflection, refraction, transmission and shadow. Ray tracing is able to produce more vivid and realistic images than typical scanline* rendering method. However, it also requires higher computational costs. As a result, ray tracing is better suited for application fields such as movies and TV programs where photorealistic images may be produced regardless of long rendering time, than for real-time applications such as video games where rendering speed is important.

* Scanline algorithm – an algorithm used to fill up the internal space of a triangle on a row by row basis.

The algorithm calculates the color of an object by casting a ray towards an object, passing through a pixel in a screen. Each ray must undergo intersection test with all the objects it may collide in the scene. Once the ray identifies the nearest object, the algorithm performs lighting* calculation for the incoming light at the point of intersection and applies the material properties of the object to decide the preliminary color of the pixel. In addition, secondary rays† are generated, depending on the reflective or refractive properties of the object, and the final color of the pixel is decided based on all the information gathered from the intersections between the primary and secondary rays, and the object.

Ray tracing algorithm assumes that rays are emitted away from the camera rather than into it. And this compares with how actual light works in reality. Assuming rays moving away from the camera is more efficient than tracing the rays casted from a light source, because forward simulation requires significant computational efforts to calculate the light path, given that the majority of rays from a light source do not travel directly into the camera. One of the typical examples of this forward simulation is photon mapping.

A simple way used in ray tracing is to update the value of a pixel after calculating the ray for the maximum number of reflection or searching for its certain travel distance without any intersection, assuming that a given ray intersects view frame. The value of a pixel can be calculated by various types of algorithms including traditional rendering algorithm and radiosity algorithm.

a. Ray

A ray is an imaginary line that travels from a designated point in certain direction. In ray tracing rendering, there are two types of rays: primary ray and secondary ray. Primary ray is a ray emitted from the point to decide the initial value of a pixel. All rays besides primary rays are called secondary rays.

b. Primary Ray

Primary ray is a ray generated from a camera to locate an object which will be displayed on a screen. Secondary ray is generated based on the information on the intersection point between the ray and the object, as well as the material property of the point. Also, primary ray is mainly responsible for shaping the outline of the object.

Since each pixel is assigned with a single primary ray which decides its preliminary color, primary ray is generated as many as the total number of pixels in a screen. Therefore, primary ray can be

* Lighting – an algorithm, also known as illumination algorithm, which calculates the value of a pixel using light sources, and the angles of incidence and reflection. Phong illumination is the most typical example and more suited for real-time rendering than others.
† Secondary ray – rays generated to express the effects of reflection, refraction, transmittance and shadow.

considered as a ray generated solely for locating the object which will be rendered on the screen. Depending on the material properties of an object and the location of the intersection points between the primary ray and the object, additional rays, defined as secondary rays, are generated.

c. Secondary Ray

All rays that are derived from primary rays are secondary rays including shadow, reflection and refraction rays.

Shadow rays express shadow effects by identifying an occluding object in between the light source and hit point. If no occluding object is identified, shadow rays are responsible for gathering the information needed for shading calculation for the light source.

Reflection and refraction rays are additional rays generated to express reflection and refraction effects depending on the material properties of an object.

1.2 Ray Tracing with RayCore® 1000

RayCore® 1000 is a ray tracing accelerator for 3D graphics rendering, which is equipped with a fully hardwired pipeline algorithm. Typical data processing for ray tracing pipeline is shown below in Figure 1. This algorithm requires significant computational time and costs due to its nature of processing each unit in a chronological order and calculating all the rays including primary ray and secondary ray in each unit. The pipeline consists of six units including ray generation unit, traversal unit, intersection test unit, hit-point calculation unit, shading unit and texture mapping unit. A ray generated towards the surface of a screen in ray generation unit explores the surface of the nearest object in traversal unit. In intersection test unit, the ray is further tested for intersection with objects, followed by the calculation of an accurate intersection point between the ray and the objects in hit-point calculation unit. A color of the intersection point is determined in shading unit, and the texture color of the intersection point is assigned in texture mapping unit. The last two units are sometimes integrated for a simpler process.

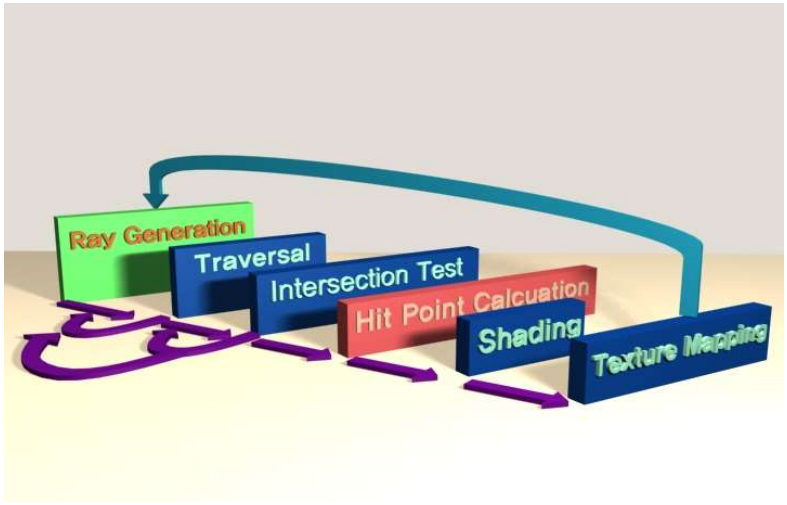

Figure 1. Typical ray tracing pipeline

As mentioned above, primary rays are emitted from a pixel in a screen, which may be further dissipated into secondary rays including shadow rays, reflection rays and refraction rays. Additional reflection and refraction occur as the secondary rays hit the surface of another object. The serial derivation of the additional rays makes ray tracing work under a software system named "recursive procedure call," and this has been a major bottleneck that has challenged the algorithm's real-time rendering process.

The hardwired pipeline that enables the real-time rendering capability of RayCore® 1000 is shown in Figure 2. T&I unit is designed to combine traverse unit and intersection test unit so that the efficiency of each function is improved. In addition, RayCore® 1000 has multiple T&I units in parallel structure and is unique in that it processes different data concurrently by using MIMD architecture.

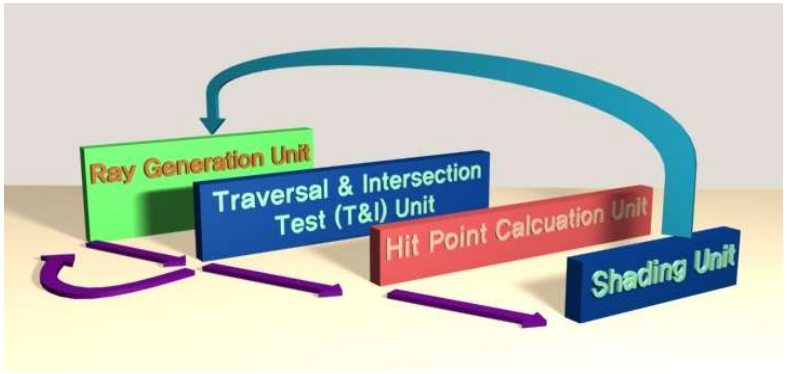

Figure 2. RayCore® 1000 pipeline

The hardware pipeline for RayCore® 1000 consists of ray generation unit, T&I unit, hit-point

calculation unit and shading unit. T&I unit can be designed in multiple units and process different data concurrently in parallel structure. Ray generation unit casts primary rays and secondary rays for all the coordinates in a given screen block. T&I unit navigates objects that have been identified by a ray and examines if any intersection between the ray and the objects occurs. If an intersection is observed, primitive data of the intersection point is transmitted to hit-point calculation unit which calculates the point of intersection, or otherwise the whole process starts from the beginning for the next coordinate. When T&I unit detects intersection, it transmits data to ray generation unit based on the material property at the intersection point. Then, ray generation unit emits second rays such as refraction rays or reflection rays which will go over the traverse process and the intersection test. Shading unit utilizes Phong illumination model when deciding the color of the intersection point calculated by hit-point calculation unit. When texture mapping has been implemented, the texture coordinate and the texture data of the intersection point is used for texture mapping. Because RayCore® 1000 is amenable to screen segmentation, setting up the system in parallel structure makes it support above Full HD level resolution.

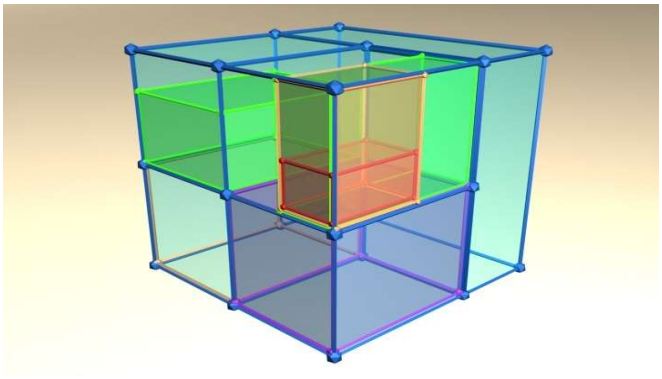

Figure 3. KD-Tree in a 3 dimensional space

a. Acceleration Structure

The application of acceleration structure is necessary for real-time ray tracing rendering, given that searching for a suitable triangle is a critical factor in determining rendering performance. Acceleration structure supports triangle searching process to make it faster and hence reinforces improved rendering performance.

KD-Tree and Bounding Volume Hierarchy(BVH) are typical acceleration structures used in ray tracing. Even though they both demonstrate high speed and efficient performance in triangle searching, RayCore® 1000 uses KD-Tree since it performs more efficiently in terms of rendering speed.

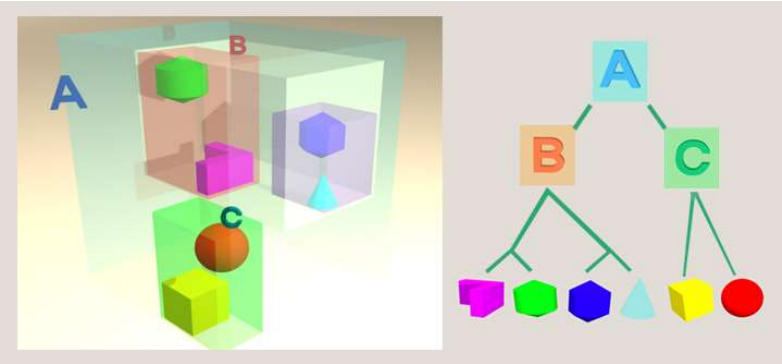

Figure 4. BVH Diagram

KD-Tree is structured by Binary Space Partitioning*. When a plane is partitioned into two different spaces, a triangle may also be partitioned concurrently. In this case, the triangle can be shared by both spaces. Unlike KD-Tree, BVH uses Binary Object Partitioning†. KD-Tree enables higher rendering speed; however, it takes longer time for building acceleration structure than does BVH.

Rasterization directly sends a triangle to the renderer instead of implementing acceleration structure in rendering process. Hence, no function exists to generate acceleration structure in rasterization. However, in ray tracing, there exists a certain stage where the algorithm needs to start creating acceleration structure. Since this stage cannot be automatically identified by API, ***rcFinish()*** should be called. After this function signals that all triangles are loaded, building acceleration structure begins.

1.3 RayCore® 1000 API

RayCore® 1000 API(Application Programming Interface) supports functions that are required in programming ray tracing contents rendered by RayCore® 1000. Through this API, 3D modeling data is transmitted to the renderer‡ which processes the data to generate ray traced images.

RayCore® 1000 API is designed familiarly to OpenGL ES 1.1. Also, the functions and parameters are similarly defined as those in OpenGL ES 1.1. However, since the rendering method of ray tracing largely differs from that of rasterization, a few differences exist in between the two types of API.

Similarities and differences which RayCore® 1000 API has in comparison to OpenGL ES 1.1 are the following:

* Binary Space Partitioning – classifies objects in a space subdivided into different spaces.
† Binary Object Partitioning – classifies spaces in an object subdivided into different pieces.
‡ Renderer – a software or hardware device that generates images.

- Functions and programming methods are similar.
- Projection transformation is not used.
- Static or dynamic status of objects can be specified.
- When loading data is complete, the commencement of rendering should be identified.

In order to set the information of textures* and material properties†, each object is activated by binding its own name. RayCore® 1000 API uses prefixes 'RC' for data types and 'rc' for functions. The way that modeling data is set by RayCore® 1000 API is equivalent to how it is set in OpenGL ES 1.1.

In OpenGL, a viewpoint can be changed by utility library functions such as glut, implying that every data can be transformed into normalized space. In RayCore® 1000 API, however, images are generated with an object fixed at certain position and with a viewpoint moving relative to the object. In addition, project transformation is avoided to eliminate every reloading of data. This way, static and dynamic objects are classified. Once static objects are loaded in the beginning, they may be reused during the rendering process without reloading. Dynamic objects, however, should be reloaded for every frame with different acceleration structures created correspondingly. This process does not exist in OpenGL. Even though reloading data for every frame requires high computational complexity, backward rendering in ray tracing is able to realize more efficient data processing even with the high computational complexity.

To determine the final color of a pixel and the types of ray tracing effects to be expressed, material properties and textures must be set. When loading data is completed, RayCore® 1000 API sends a signal to RayCore® 1000 to commence rendering process.

a. Programming Model

RayCore® 1000 API requires the following two stages in application programming:

- Initial setting for screen
- Loading rendering data

Initial settings for screen such as frustum and background color indicate data or setting that does not change with frames. Sometimes, the setting for light sources can be considered as the initial setting given that the setting for light sources stays constant. Also, this setting may be used in loading rendering data, because the decrease in processing speed it may cause is trivial. On the other hand, the setting for texture data should always be in the initial setting for screen. Technically, it is plausible to change data for texture; however, it leads to diminishing speed of rendering speed due to

* Texture – image data used in rendering with texture mapping method.
† Material Property – material property that creates effects such as reflection, refraction and transmittance in addition to ambient, diffuse and specular effects from Phong shading.

the time taken during the data conversion. To avoid the decline in rendering performance, texture data is loaded in the initial stage and reused during the following process.

If the material information used with texture data is set together, the usage of the material and texture becomes easier. Setting the texture as one of the material properties is more efficient than setting the texture and material separately. Therefore, it is strongly recommended to specify the material in process of initial setting after loading the texture data.

Rendering data which contains dynamic data means that data is changed for every frame. The rendering model data is changed every time, because the model is moved according to the user input or the determined order. Even without any movement, this model data should be set continuously for rendering. This process is repeated until the rendering application program is terminated. The texture coordinates of the model is regarded as rendering data. If the light source and the view point are changed, new data should be set all the time.

b. Object Naming

Certain data for determining its texture and material property is set to an object which is assigned with a specific name. First, texture object and material object are created. Then, each object is assigned with a name in numbers and delivered to application programs. Using the name, the object is activated, and its data is set. The texture and material may be specified in the following order:

1. Each object is created.

2. Among the created objects, an object is selected with a name which is granted during the creation of the objects.

3. Data is loaded in the selected object.

4. When an object is activated by calling its name, the data loaded in the object can be used.

1.4 Terminology

Major terminologies used in RayCore® 1000 Programming Guide are defined as the following:

- **Model** – 3D data to be rendered into images.
- **Object** – a set of triangles. A group of objects and triangles or a group of objects can also be defined as an object. The minimum unit of an object is a set of triangles containing one material property. An object in the minimum unit has only one material property. If an

object has more than one material property, the object is divided into as many objects as the number of material properties to compose an object in upper object class.

- **Triangle** – a polygon with an area composed of three vertices. Data type used in RayCore® 1000 rendering is in triangle.

- **Primitive** – a primitive polygon used in the rendering process of RayCore® 1000. Primitive is a native terminology for RayCore® 1000 and equivalent to triangle.

- **Texture** – image data expressing the texture of an object.

- **Material Properties** – Properties of a material. Properties of the color and texture of an object. Properties of reflection, refraction and transmittance. A set of properties that respond to lights.

Chapter 2 Viewing

Viewing an object requires various types of information among which view point and viewing volume may be typical examples. As one of the most basic elements in viewing, view point indicates the information of a viewer such as an imaginary eye or camera. Viewing volume, on the other hand, implies the quantitative value of a region of space in which an object can be seen. This chapter explains what roles view point and viewing volume play in and how their settings can be applied to 3D graphics. In addition, the results of data processing for view point and viewing volume will be covered in this chapter.

2.1 View Point

View point has the information needed to observe an object in a 3 dimensional space. The information includes the location of a viewer as well as the direction and the region of viewing, all of which allow the object to become visible. The following functions are called to provide the settings for view point.

void **rcuLookAt** (RCfloat *eyex*, RCfloat *eyey*, RCfloat *eyez*, RCfloat *centerx*, RCfloat *centery*, RCfloat *centerz*, RCfloat *upx*, RCfloat *upy*, RCfloat *upz*);
Coordinates of the view point and the reference point and *UP* vector are set. The position of a view point indicates the coordinate of the view point in a 3 dimensional space. The reference point indicates the center of screen, and is used to calculate a vector moving from the view point to that reference point. *UP* vector assigns the elements of the vector, indicating which direction is upward and perpendicular to the view point.

Example 1

rcuLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);

The coordinate of the view point and the center of screen are (0, 0, 0) and (0, 0, -1) respectively. And, the *UP* vector is heading towards the positive direction in the *y*-axis with a unit magnitude. This is a default setting in RayCore® 1000 API and will not be changed unless new setting is made.

Example 2

rcuLookAt(1, 1, 1, 0, 0, 0, -1, 1, -1);

The coordinate of the view point and the center of screen are (1, 1, 1) and (0, 0, 0) respectively. And, the *UP* vector is heading towards the coordinate of (-1, 1, -1). Once view point is set, setting the viewing volume must be followed.

2.2 Viewing Volume

Viewing volume implies the region of a space where an object is present to appear in the display. Viewer's position and viewing direction are determined by view point, whereas the space in which objects are present is set by viewing volume, also known as view frustum. The visual field is divided into two different spaces where an object can be seen in one space but is not detected in the other.

a. Resolution

Images are created by rays emitted within viewing volume. Resolution, expressed by the number of pixels composing a single screen, determines the number of rays generated. Resolution can be set by calling ***rcViewport()***.

void **rcViewport** (RCint *x*, RCint *y*, RCsizei *width*, RCsizei *height*);
rcViewport() Sets resolution. The resolution verifies the number of pixels in a screen and hence the number of primary rays generated. The current version of RayCore® 1000 API does not use x and y values. The width and height of final images are set by *width* and *height* respectively.

Default resolution is 800x480. If ***rcViewport()*** is not used, images are created in the default resolution. In addition to determining the number of primary rays, this default resolution decides the initial direction of the rays and calculates the variation in the emission angles.

b. Frustum Setting

Frustum indicates the region of which an object will be shown based on the viewing direction. ***rcFrustum()*** and ***rcuPerspective()*** set the information of frustum.

i. Direct Frustum Setting

rcFrustum() is a function that directly sets the information of frustum.

void **rcFrustum** (RCdouble *left*, RCdouble *right*, RCdouble *bottom*, RCdouble *top*, RCdouble *zNear*, RCdouble *zFar*);
rcFrustum() appoints the size of clipping plane expressed by *left*, *right*, *bottom* and *top*. Also, this function defines the distance from the view point to the clipping planes in long and short distance. *zFar* is not used.

The values specified by above functions are used to calculate the direction of the first ray. The directions of next rays are calculated by the difference in emission angles, and this calculation needs to be performed in the beginning of initial screen setting. The values can be reused unless view frustum is changed.

rcFrustum (-1, 1, -1, 1, 1, 1000);

Above code demonstrates the example of setting the size of a clipping plane in one unit distance. The width and height of the plane is one unit each, and the near distance is one unit. The value for the long distance among the above parameters is not used in ray tracing.

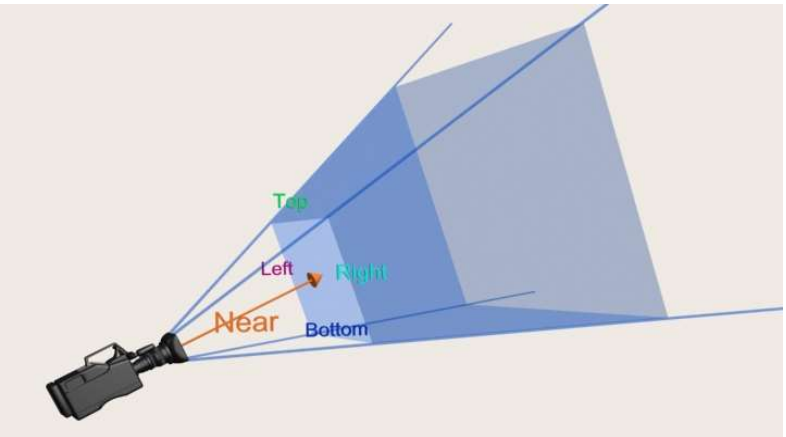

Figure 5. View frustum

ii. Aspect ratio Setting

Since ***rcFrustum()*** sets the region of frustum with actual values, final images may be seen distorted when the aspect ratio changes. Hence, ***rcuPerspective()*** is used to fix the aspect ratio of frustum and avoid the distortion.

void **rcuPerspective** (RCfloat *fov*, RCdouble *aspect*, RCfloat *zNear*, RCdouble *zFar*);
Vertical *fov*(field of view) and *aspect* value are set. *fov* and *aspect* value derive the width of frustum. *zNear* and *zFar* are equivalent to the parameters of rcFrustum function.

The field of view *fov* indicates the viewable region with degree of angle, and its range is in between 0˚ and 180˚ in vertical axis. *fov* provides the information to derive the height and the aspect ratio of frustum, with which the width of the frustum can be calculated.

aspect indicates the ratio of width relative to the height of frustum. Frustum with a fixed aspect ratio can keep the constant ratio between the screen and the object.

RayCore® 1000 API does not create projection matrices when setting the values for *fov* and *aspect*. In general, rasterization method projects all objects into a normalized space through projection matrices. These projection matrices are used together with transformation matrices when modeling coordinate system is converted into the world coordinate system. However, this process is not used in ray tracing.

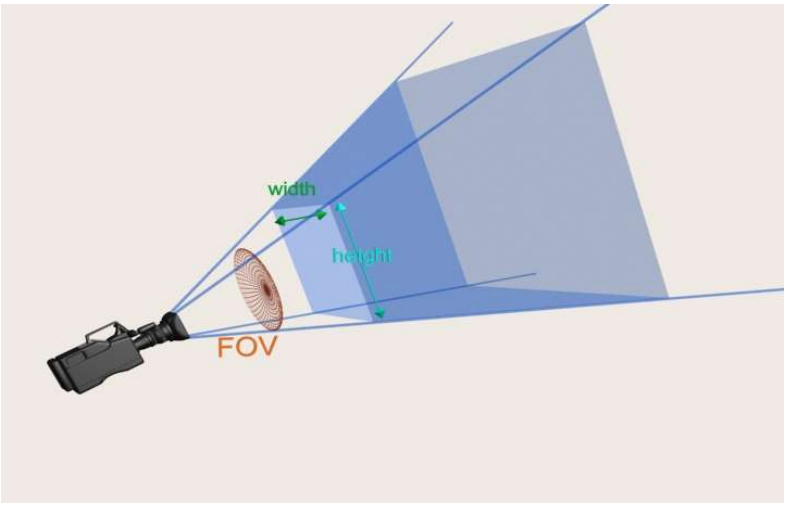

Figure 6. Frustum with *fov* and *aspect*

2.3 Initialization

The most important settings that need to be initialized in ray tracing are the settings for view point and frustum. Initialization can be executed by the functions explained above.

2.4 Unnecessary Projection Matrix

In order to generate 3D graphics images, the coordinates of the vertices for triangles in a three dimensional space are needed. Matrices are useful to efficiently manage the coordinate information. When the coordinate information is transformed using matrices, an object is relocated. These coordinate transformation matrices can also be applied to a three dimensional space.

Transforming the coordinates into the world coordinate system works the same in both rasterization and ray tracing. However, projection transformation is performed differently. In rasterization, **PROJECTION_MATRIX** mode enables processing for projection transformation under rasterization method. Projection matrices set in that mode relocate all objects to a normalized cube. Only objects inside the cube are rendered. The space where an object is seen is determined by viewing volume.

Ray tracing method does not perform projection transformation. Instead, projection effects are expressed by the generation of rays. Objects are projected by generating rays emitted towards each pixel in a screen. This way, ray tracing produces effects equivalent to those created by rasterization that uses projection matrices.

2.5 Sample Codes

a. Frustum by FOV

Basic settings are the following:

- Setting for background color
- Setting for screen resolution
- Setting for frustum by FOV
- Setting for the position of view point and the direction of viewing

Sample codes are the following:

```
{
  rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
  rcViewport(0, 0, 800, 480);
  rcuPerspective(60.0f, 800.0f / 480.0f, 0.1f, 100.0f);
  rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);
}
```

Background color setting for the red, green and blue are 0.73, 0.2 and 0.23 respectively. *fov* is fixed at 60˚, and the ratio of the screen is set at *aspect*. The screen is distant from the view point by 0.1, and the maximum range is set to 100.

The rendering result is shown in figure 7. Programming method to draw the triangle will be explained in a later chapter. Since frustum is set by *fov*, the shape of the triangle is not distorted.

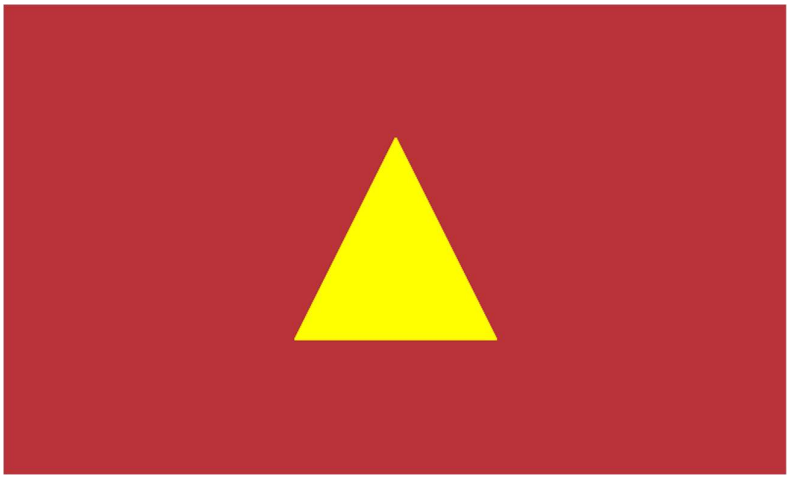

Figure 7. Standard triangle drawn with *fov* setting

b. Frustum by Values

Below is the setting for frustum based on actual values.

- Frustum is set based on actual values.

Sample codes are the following:

```
{
  rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
  rcViewport(0, 0, 800, 480);
  rcFrustumf(-0.1, 0.1, -0.1, 0.1, 0.1, 100);
  rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);
}
```

The screen is distant from the view point by 0.1. Top and bottom value are set to 0.1 and -0.1 respectively. Left and right value are set to -0.1 and 0.1 respectively. Using the ratio between the width and height of the screen gives an equivalent result that may be produced with frustum set by *fov*.

Below is the rendering result with frustum set by values. This setting ignores the fixed ratio between the height and width of the screen, and therefore the image looks distorted relative to the image produced by using ***rcPerspective***.

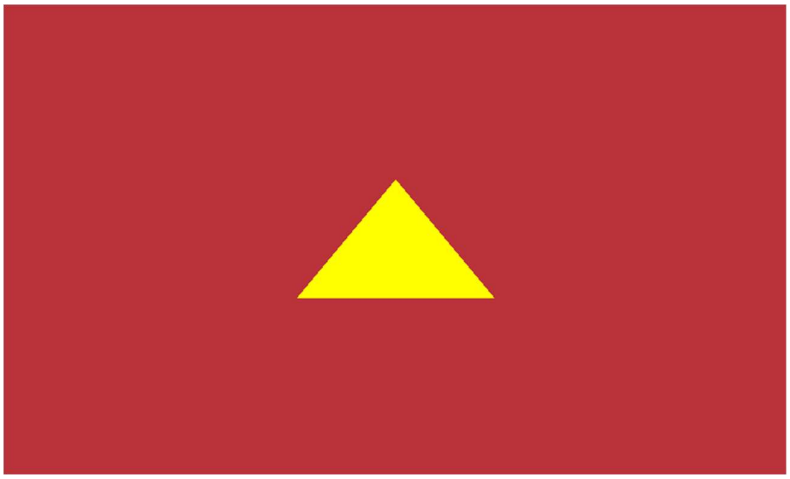

Figure 8. Standard triangle drawn with frustum setting by values

c. Viewpoint Position

The position of a viewpoint can be changed by calling ***rcuLookAt()***.

- Transform the position of a viewpoint

Sample codes are the following:

```
{
  rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
  rcViewport(0, 0, 800, 480);
  rcuPerspective(60.0f, 800.0f / 480.0f, 0.1f, 100.0f);
  rcuLookAt(0.3, 0, 1, 0, 0, 0, 0, 1, 0);
}
```

The viewpoint is shifted to the x-axis by 0.3. The viewing direction stays the same. The rendering result is shown below.

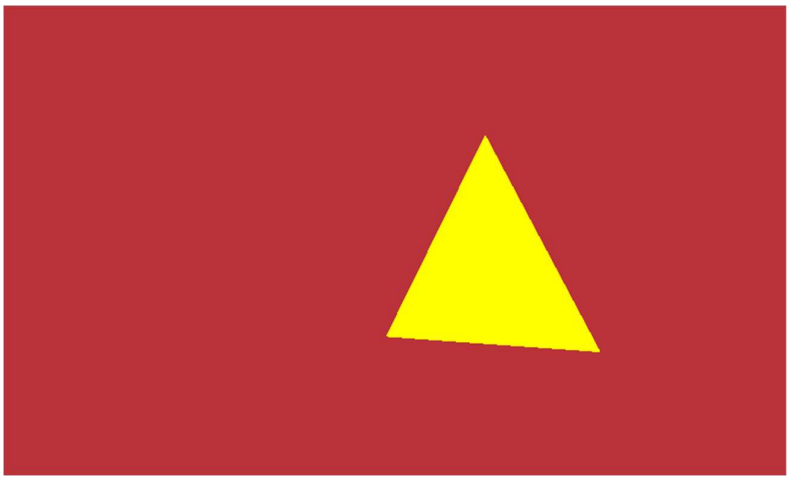

Figure 9. Standard triangle with the changed position of the viewpoint

Chapter 3 Transformation

An individual object is composed in its own model coordinate system. And, a group of the objects is transformed into the world coordinate system. This transformation is achieved by transforming the coordinates of vertices. In addition, transformation is applied to the translation of objects.

RayCore® 1000 API supports functions that are used in transforming the coordinates. There are three types of coordinate transformation: translation, rotation and scaling. Each transformation is performed by matrix multiplication, and a matrix is structured in 4x4 to allow affine transformation.

3.1 Model Transformation

Sixteen elements in a matrix are arrayed in column major order.

$$M = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

a. Translation

Translation of an object depends on the direction and magnitude of the object's movement. The direction of translation is determined by axes, and the amount of movement is expressed in magnitude.

Below is a sample translation matrix:

$$\mathrm{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following function executes translation:

void **rcTranslatef** (RCfloat *x,* RCfloat *y*, RCfloat *z*);
An object is translated by *x*, *y*, *z* units about each corresponding axis.

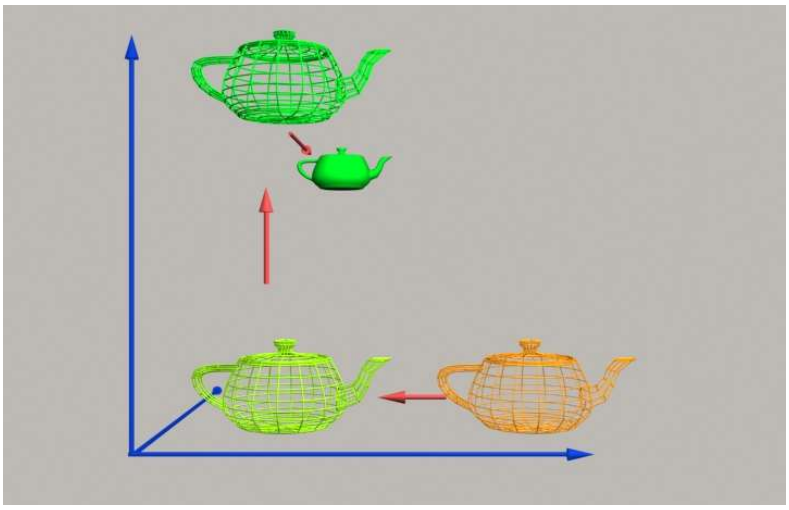

Figure 10. Translation

b. Rotation

Rotation requires the information of how much an object will be rotated to which direction. Rotation axis cannot be randomly chosen. One of the axes in the coordinate system should be selected as a base axis.

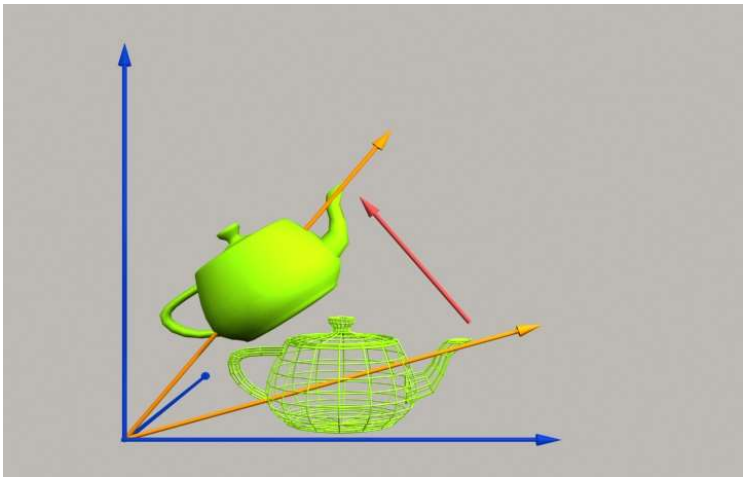

Figure 11. Rotation

Sample rotation matrices are shown below. R_x, R_y and R_z indicate the rotation matrix in x-axis, y-axis and z-axis respectively.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following function executes rotation:

void **rcRotatef** (RCfloat *angle*, RCfloat *x*, RCfloat y, RCfloat *z*);
Base axis for rotation is set by *x*, *y* or *z*. This function executes the rotation of an object by *angle* ° about the corresponding axis in counter-clock wise direction.

c. Scaling

Scaling modifies the size of an object. Scaling multiplies a coordinate by certain value.

Below is the example of a scaling matrix:

$$s = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following function executes scaling:

void **rcScalef** (RCfloat *x*, RCfloat *y*, RCfloat *z*);
An object may be enlarged or condensed, depending on the input value. (*x*, *y*, *z*) indicates an input value that changes the size of an object about the origin.

If the object stays off the origin, the scaled object is not enlarged or condensed in exact proportion to the input value; instead, it is scaled in proportion to the distance between the object and the origin.

d. Matrix Transformation

When users create their own model matrix in programming applications, model transformation matrix can be modified by corresponding matrix data. In addition, matrix multiplication can create totally new matrix by applying the combination of the above transformation functions. Newly created matrix may be assigned to the existing matrix or multiplied by the existing model transformation matrix to become new model transformation matrix. Final model matrix can be confirmed by calling ***rcGet()***.

void **rcLoadMatrixf** (const RCfloat **m*);
Matrix data is loaded. Matrix data *m* indicates the pointer value representing the data of a matrix which is stored in column major order. Data size should be represented by 16 elements, and this data replaces the data for the existing model transformation matrix.
void **rcMultMatrixf** (const RCfloat **m*);
The input matrix data *m* is equal to that of the above. By multiplying this data with the existing model transformation matrix, a new matrix is created.

e. Push / Pop Matrix

Final transformation matrix is composed by the multiplications of various matrices that are created by multiple transformations. In order to undo the multiplications, inverse matrix is multiplied. However, forming a 4x4 matrix for the multiplications of the inverse matrix is inefficient. Therefore, original matrix is saved in memory and loaded back whenever needed. This way, recalculation of the matrix is avoided. Matrix stack allows the process of saving and loading the matrix, which is

controlled by ***rcPushMatrix()*** and ***rcPopMatrix()***.

f. Matrix Stack

Transformation can easily apply to an object model in hierarchical structure such as in solar system with the push/pop method. The solar system is a good example. Earth rotates around the Sun, and the Moon rotates around Earth. If the Sun is not in the origin of the coordinate system, the Sun should be relocated to the origin, and Earth starts to rotate around the origin. And this explanation also applies to the relationship between Earth and the Moon. However, if Mars is introduced in the example, it gets more complicated. One way to resolve this is to change the position of the Sun and let the Mars rotate around the origin. And then, a translation matrix that brings them back to the original position may be multiplied with a rotation matrix that allows the rotation of the Mars. The other method is to reuse matrix data stored in stack memory. The first method is to create new transformation matrix using the functions explained above. And the second method is to store the current matrix in stack memory and load back by calling the functions below when needed.

void **rcPushMatrix** (void);
Current transformation matrix is stored in stack memory.

rcPushMatrix() stores the current settings for the matrix and load back when needed.

void **rcPopMatrix** (void);
Matrix saved in stack memory is loaded as transformation matrix.

Transformation restarts from the previous matrix.

3.2 Sample Codes

a. Translation

Sample codes for translation are the following:

```
#include "RCFramework.h"

// Programming Guide - Translate

struct Pos {
  float x, y, z;
};
```

```
RCubyte g_cubeIndices[] = {
  0, 1, 2,
};

struct Pos g_TrianglePos[] = {
  {0.0f, 1.0f, 0.0f},     // 0
  {-1.0f, -1.0f, 0.0f},   // 1
  {1.0f, -1.0f, 0.0f},    // 2
};

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Translate", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcTranslatef(1.0f, 1.0f, 0.0f);
        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1,1,0,0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f );
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }
```

```
  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
  }
};

Tutorial      g_Tutorial;
```

The object is translated by 1 to the positive x-axis and 1 to the positive y-axis. The triangle is moved from the center as shown below.

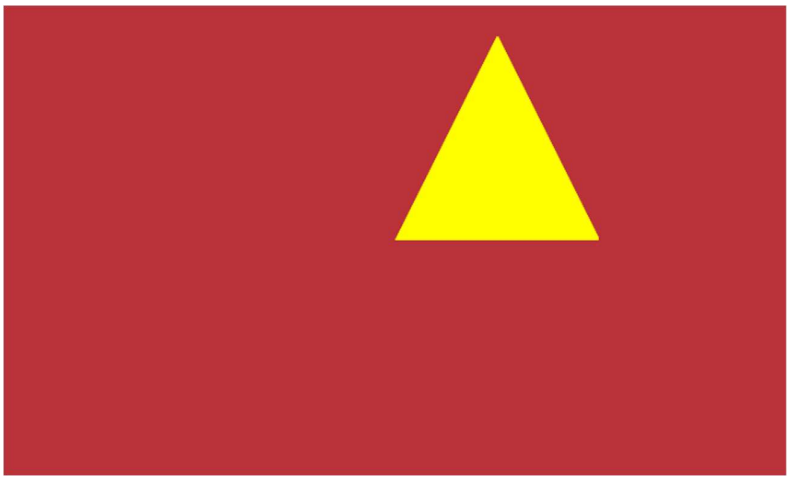

Figure 12. Translation

b. Rotation

Sample codes for rotation are the following:

```
#include "RCFramework.h"

// Programming Guide - Rotate

struct Pos {
  float x, y, z;
};

RCubyte g_cubeIndices[] = {
  0, 1, 2,
};
```

```
struct Pos g_TrianglePos[] = {
  {0.0f, 1.0f, 0.0f},     // 0
  {-1.0f, -1.0f, 0.0f},   // 1
  {1.0f, -1.0f, 0.0f},    // 2
};

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Rotate", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcRotatef(45, 0, 0, 1);

        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(),0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }

  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
```

```
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
    }
};

Tutorial      g_Tutorial;
```

The triangle is rotated by 45˚ about z-axis as shown below.

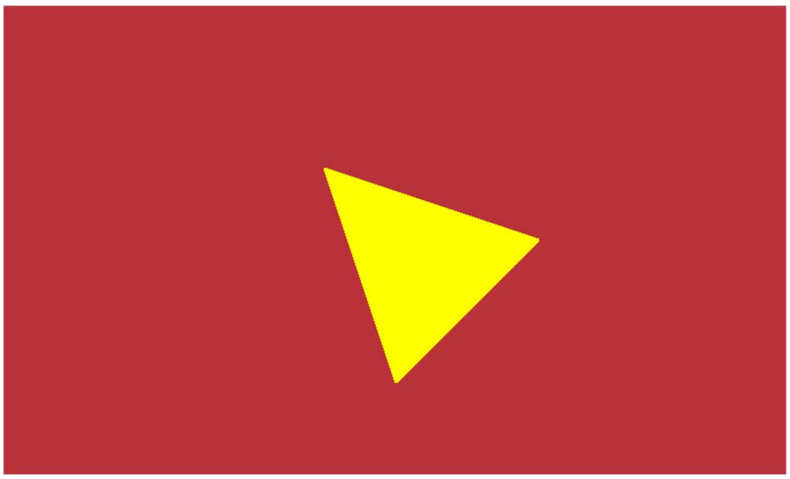

Figure 13. Rotation

c. Scaling

Sample codes for scaling are the following:

```
#include "RCFramework.h"

// Programming Guide - Scale

struct Pos {
    float x, y, z;
};

RCubyte g_cubeIndices[] = {
    0, 1, 2,
};

struct Pos g_TrianglePos[] = {
    {0.0f, 1.0f, 0.0f},      // 0
    {-1.0f, -1.0f, 0.0f},    // 1
    {1.0f, -1.0f, 0.0f},     // 2
};
```

```
class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Scale", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcScalef(2.0f, 2.0f, 1.0f);

        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }

  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
```

```
  }
};

Tutorial      g_Tutorial;
```

The size is enlarged twice the original size about x-axis and y-axis as shown below:

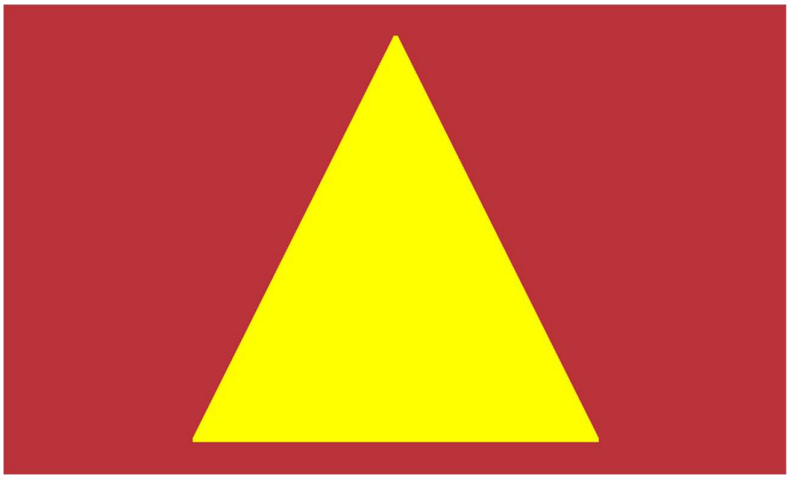

Figure 14. Scaling

If the input value is less than 1, the triangle is condensed. If the input value is 0, there will be no triangle displayed since all the vertices will be moved to the origin of each axis.

d. Mixed Transformation 1 (Translation After Rotation)

Translation after rotation and rotation after translation provide different results. Below is the example of translating a triangle by 1 towards the positive side of x-axis after rotating by 90˚ about z-axis. Here, the transformation matrices are multiplied to the right side of the base matrix.

```
#include "RCFramework.h"

// Programming Guide - Transform(Rotate, Translate)

struct Pos {
  float x, y, z;
};

RCubyte g_cubeIndices[] = {
  0, 1, 2,
};

struct Pos g_TrianglePos[] = {
  {0.0f, 1.0f, 0.0f},     // 0
  {-1.0f, -1.0f, 0.0f},   // 1
  {1.0f, -1.0f, 0.0f},    // 2
```

```
};

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Transform(Rotate, Translate)", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();


        rcTranslatef(1, 0, 0);
        rcRotatef(90, 0, 0, 1);

        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }

  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;
```

```
        DynamicSceneDraw();

        return TRUE;
    }
};

Tutorial      g_Tutorial;
```

The final image is rendered as shown below.

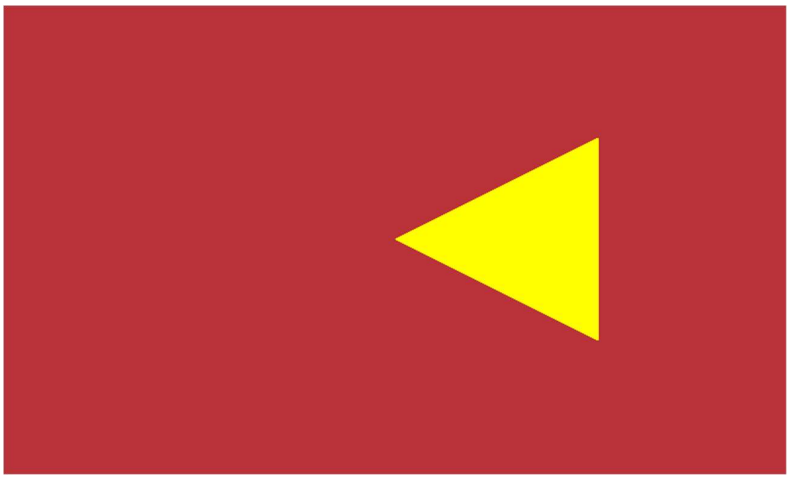

Figure 15. Mixed transformation 1 (translation after rotation)

e. Mixed Transformation 2 (Rotation After Translation)

Translation of a triangle is performed first, followed by rotation. The resulting image looks different from that of the previous example.

```
#include "RCFramework.h"

// Programming Guide - Transform(Translate, Rotate)

struct Pos {
    float x, y, z;
};

RCubyte g_cubeIndices[] = {
    0, 1, 2,
};

struct Pos g_TrianglePos[] = {
    {0.0f, 1.0f, 0.0f},     // 0
    {-1.0f, -1.0f, 0.0f},   // 1
    {1.0f, -1.0f, 0.0f},    // 2
};
```

```
class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Transform(Translate, Rotate)", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();


        rcRotatef(90, 0, 0, 1);
        rcTranslatef(1, 0, 0);


        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f );
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }

  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();
```

```
        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

The triangle is first translated by 1 towards the positive direction of x-axis and then rotated by 90˚ about z-axis. The resulting image looks the same as the image that would have been generated if the triangle is translated by 1 towards the positive side of y-axis after being rotated by 90˚ about z-axis.

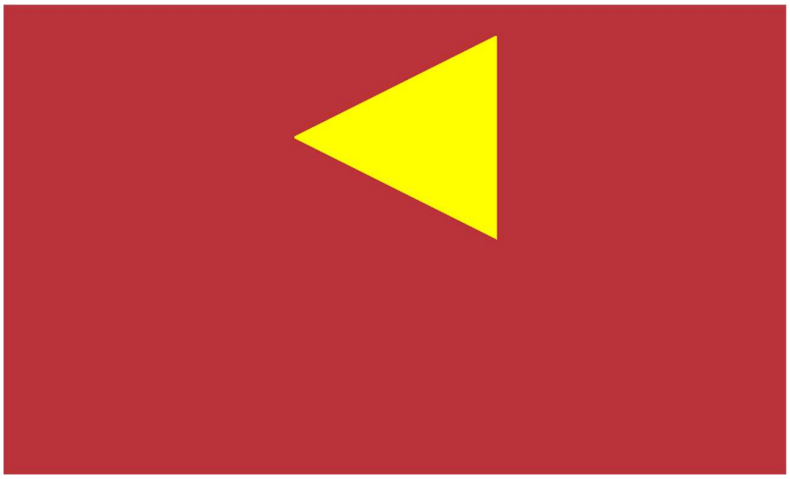

Figure 16. Mixed transformation 2 (rotation after translation)

As shown in the mixed transformation 1 and 2, final images largely depend on the order of transformations.

Chapter 4 Light Source

Light is another major subject in ray tracing rendering. If lights do not exist, objects cannot be identified. This chapter explains the settings of light source and their implications.

4.1 Light Property

The following is the list of light properties:

- Position
- Type
- Ambient property
- Diffuse property
- Specular property
- Direction vector
- Spot light*

a. Position and Type of Light Source

The position of light source can be defined as (x, y, z, w), where (x, y, z) means the position in a 3 dimensional space and w indicates the type of light source.

Light source can be either fixed at certain position or changed in every frame. In the situation where its position changes with every frame, changing the position can be made by setting new input values or using a coordinate transformation function. Since light source is considered as a rendering object, applying transformation matrix to light source is valid.

* Spot light – a light that focuses on a small area to increase the level of concentration.

Light source can be divided into two different types: point light source and directional light source. The sun in the solar system is a good example of point light source which is a type of light source that emits light into random directions. Lighting calculation is performed with an arbitrary direction vector originated from the position of light source.

On the other hand, directional light source has its unique direction. Instead of the position of light source, a direction vector is used in lighting calculation.

b. Light Property

For lighting calculation, a certain property is assigned to light source. Ambient, diffuse and specular are the examples of light properties. Each of them specifies the material property caused by light. Further details will be explained in later chapter after material property is covered.

c. Spot Light

Spot light is a unique example of point light source. Focusing on small part of the area with its unique position and direction, the light is expressed in a cone shape.

Figure 17. Spot light

d. Attenuation

When light source is far away from an object, light can be attenuated by distance. In this case, the intensity of attenuation can be specified. The following three are the attenuation types:

- Constant attenuation
- Linear attenuation
- Quadratic attenuation

e. Programming

The following is the list of light source settings:

- Light source can be used only if lighting is enabled.
- Default light source is used when lighting is disabled or separate light source is not set.
- The maximum number of light source is 8.
- On/off setting is applied to individual light source.

Light source setting is made by the following settings:

void **rcLightv** (RCenum *light*, RCenum *pname*, const RCfloat **params*); void **rcLight** (RCenum *light*, RCenum *pname*, const RCfloat *param*);
Light source is set by the above functions. *light* decides which light source will be activated. *Pname* assigns the properties of light source. The RGB color properties are set by RC_POSITION, RC_AMBIENT, RC_DIFFUSE, RC_SPECULAR, RC_SPOT_DIRECTION, RC_SPOT_EXPONENT, RC_INNER_CONE, RC_OUTER_CONE, RC_ATTENUATION_RANGE, RC_START_ATTENUATION, RC_END_ATTENUATION, RC_CONSTANT_ATTENUATION, RC_LINEAR_ATTENUATION, RC_QUADRIC_ATTENUATION. Input values are set by *params* or *param.*

4.2 Sample Codes

Below sample codes are used for the example in Appendix C. Textures are used to express the earth, and light source is rotated.

a. Initialization of Light Source

Light source is initialized.

```
rcEnable(RC_LIGHTING);

i = 0;

while(i < sceneData.m_lightCount) {

    RCenum lightNumber = RC_LIGHT0 + i;

    if(i < sceneData.m_lightCount) {
        rcEnable(lightNumber);

    sceneData.m_light[i].pos.w = 1;
    sceneData.m_light[i].pos.x -= 15;

    rcLightfv(lightNumber, RC_POSITION, &sceneData.m_light[i].pos.x);
    rcLightfv(lightNumber, RC_AMBIENT, &sceneData.m_light[i].ambient.r);
    rcLightfv(lightNumber, RC_DIFFUSE, &sceneData.m_light[i].diffuse.r);
    rcLightfv(lightNumber, RC_SPECULAR, &sceneData.m_light[i].specular.r);
    i++;
}
```

Each light source is activated after the setting for lighting is made. And then, the properties of each light source are set.

b. Rotation of Light Source

Light source is rotated by transformation matrices.

```
rcMatrixMode(RC_MODELVIEW);
rcLoadIdentity();

rcPushMatrix();
rcRotatef(g_turn, 0, 1, 0);
rcLightfv(RC_LIGHT0, RC_POSITION, &sceneData.m_light[i].pos.x);
rcPopMatrix();
```

If the transformation matrix is already created, the value for the position of light source and the transformation matrix are multiplied each other to determine the new position of the light source.

Chapter 5 Texture

Texture mapping is one of the necessary processes in 3D graphics to efficiently generate realistic images. This chapter introduces the functions and applications of texture mapping.

5.1 Texture Mapping

It is almost impossible to model photorealistic objects only with polygons. This is mainly because the significant number of polygons used to generate such objects requires massive computational calculations that may not be satisfied by the capabilities of current computer hardware. In addition, real-time rendering in 3D graphics should accompany the process of reducing the size of data. Instead of modeling each object in geometric shape, overlaying an image on an object can resolve the technical challenge. Also, overlaying an image allows texture expression on the object and thereby adds more realism. Texture mapping has become an essential element in the current generation of 3D graphics.

In texture mapping, the texture coordinates of vertices should be secured, and a particular image should be selected as a texture image. Images can express the texture of an object or provide visual expression in certain shapes.

The number of texture images used is limited, and an object is specified with a particular ID. Loading texture data every time significantly lowers the performance of real time processing; therefore, it should be loaded in the initialization of 3D data.

a. Texture Data

Texture data is expressed in 2D images. Image data expresses the texture of triangles or certain objects. It is almost impossible to express texture with geometric data such as triangle. Texture

mapping is essentially useful since it allows the expression of texture in addition to reducing the amount of data for real-time rendering.

Figure 18. Texture images

Using the images above leads to the simplification of modeling. Photos or realistic images are used to enhance the quality of images.

b. Texture Coordinate

In texture mapping, each vertex should be assigned with a unique coordinate which is set in modeling stage. The assigned coordinate and vertex data are loaded in a similar way. Related functions are explained in Chapter 7.

Below is an example of simple texture mapping. Basically, an image is attached to a rectangle. Figure 19 shows how each vertex is given its coordinate. Rendering results vary with the coordinate setting.

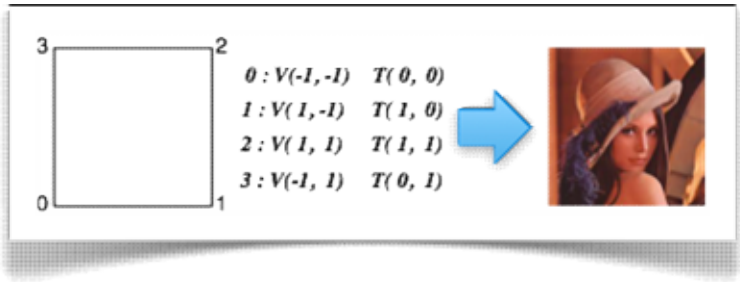

Figure 19. Texture mapping on a rectangle

5.2 Texture Filtering

When a ray hits certain point on a triangle, a texture pixel is mapped. This process is named texture mapping. The basic unit of texture pixel is called texel*. Depending on the depth or distance of a triangle, multiple texels may be used. Also, a single texel may be applied to multiple pixels on screen. In this case, aliasing effects may occur, which result in unnatural rendering outcomes. Texture filtering alleviates the problem caused by aliasing effects.

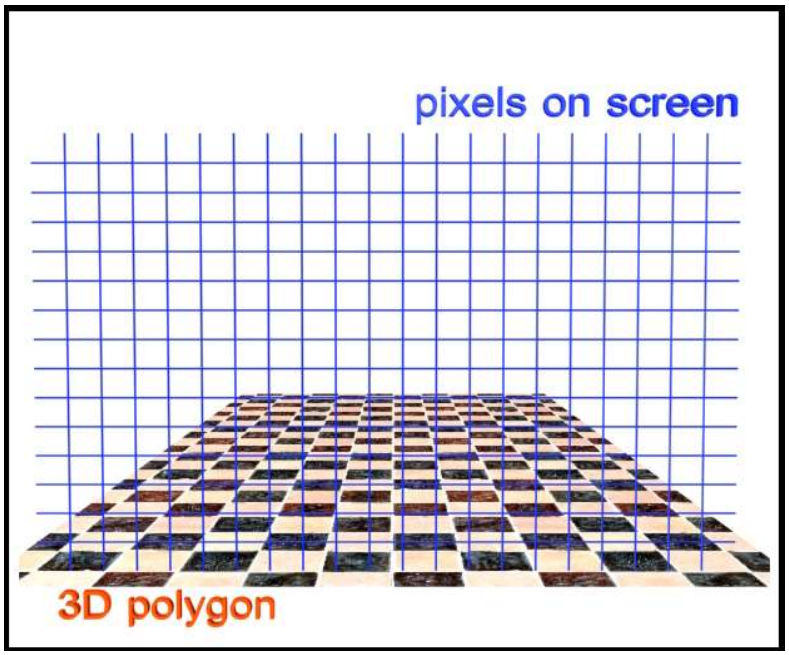

Figure 20. Each pixel with multiple texels

Applying texture filtering provides smooth rendered images. Figure 21 shows the comparison between two images where texture filtering is used for the image on left had side but not for the other. With texture filtering applied, mapping results look smoother and more natural. Among various filtering techniques, texture mipmapping is widely adopted. RayCore® 1000 API also supports texture mipmapping. Although the current version of RayCore® 1000 API supports only bilinear texture mipmapping, additional types of texture mipmapping will be supported in later versions.

* Texel – Texture pixel

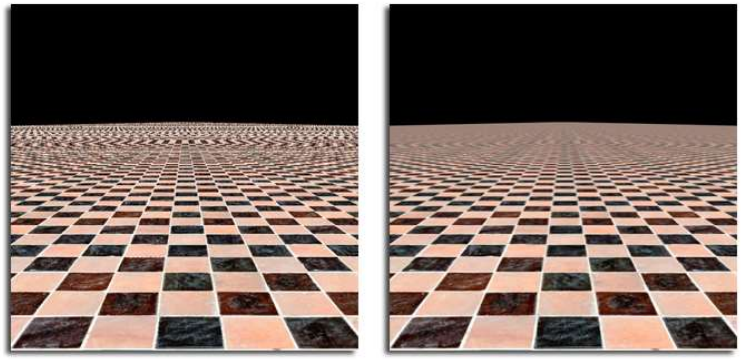

Figure 21. Filtering applied (RHS) vs Filtering not applied (LHS)

a. Mipmap Texture

The size of triangles with textures overlaid may be changed every time. Hence, the results of texture mapping may look unrealistic, mainly because the connectivity between each pixel weakens when a small texture is imposed on a large object. Using mipmap textures resolves this problem during the filtering process.

Texture mipmapping is a technique that uses the pre-store data of images after reducing the size of an image by 1/4 the original size and ranking the reduced images based on their size. Reducing the size of images every time and using a great number of texels to alleviate the connectivity lead to the decrease in rendering speed and hence inefficiency of rendering performance. In order to increase the efficiency, upper ranking images are saved. It is true that mipmapping takes a considerable amount of time; however, mipmapping process is performed prior to rendering process and therefore does not affect rendering time.

Figure 22. Texture mipmap

Texture filtering is also performed prior to rendering process. Even though it doesn't contribute much to the accuracy of images, texture filtering improves rendering efficiency. No exact values are used in terms of image quality; however, there is no disagreement that this method provides qualified performance. Therefore, using texture filtering is not optional but necessary.

5.3 Texture Objects

Textures are processed on per-object basis. When texture objects are created, the name of each object is loaded. Mapping is performed by binding a texture to this name in the list of indices.

void **rcGenTextures** (RCsizei *n*, RCint **textures*);
n number of texture objects are created. When the objects are created, the name of each object is received in *textures*. The received name can be reused continuously unless deleted. Object names are generated sequentially form 1 but not continuously. The space in *textures*, where object names are stored should be allocated prior to saving the names.

The name of a texture object is a virtually specified value that maintains texture objects used in the internal library of RayCore® 1000 API and higher levels of application programs. ***rcBindTexture()*** lets the name of texture specify which texture object is loaded or used.

void **rcBindTexture** (RCenum *target*, RCint *texture*);
Because RayCore® 1000 API supports only 2D textures, *target* must be RC_TEXTURE_2D. *texture* specifies the name of a texture object. In order to use a texture object, its name must be set.

rcTexImage2D() is the function that stores texture data. Calling this function loads the texture data to RayCore® 1000. The mipmap texture is automatically generated. The memory access pattern is pre-determined because RayCore® 1000 uses a bilinear filter internally. In other words, texels indicated by the texture coordinate (u, v) are selected by picking four adjacent texels.

Using textures requires high memory usage due to its high locality. Therefore, a cache memory is implemented in the texture mapping unit to reduce the memory bandwidth. If the texture data is rearranged according to the method of processing texture filtering, higher performance can be realized. This rearrangement of the texture data is a pre-processing step to enhance the processing performance. This process is managed by basic functions without having to specify a function to load the data.

void **rcTexImage2D** (RCenum *target*, RCint *level*, RCint *internalformat*, RCsizei *width*, RCsizei *height*, RCint *border*, RCenum *format*, RCenum *type*, const RCvoid **pixels*);
Above function loads texture images. Because only 2D textures are supported, *target* must be **RC_TEXTURE_2D**. The functionality that automatically generates texture mipmap is added in ***rcTexImage2D***. Therefore, only 0 receives a rank, followed by higher numbers automatically assigned with the corresponding mipmap. The function used in OpenGL assigns a rank to every mipmap. ***rcTexImage2D*** also determines the width and height of a texture. The size of a texture should be expressed as powers of 2. The minimum value of the size is 16, and the maximum value of the size is 1024. If the requirements of this range are not satisfied, an error occurs without loading a texture. The type of texture data supports RGB or RGBA. An internal texture always consists of the four elements of RGBA. Therefore, *format* supports RC_RGBA and RC_RGB which both specify the elements of input data. *internalformat* must be equal to *format*. *type* specifies the data type of each element. Supported data types are **RC_BYTE**, **RC_UNSIGNED_BYTE**, **RC_SHORT** and **RC_UNSIGNED_SHORT**. Data address is transferred through *pixels*.

A texture mapping unit in RayCore® 1000 consists of the 4 elements of RGBA. ***rcTextureAlpha*** specifies an alpha value added to the data with a pre-defined RGB value. If the alpha value is not defined separately by a programmer, it is set at the default opaque value of 16. (“0” indicates transparent property)

void **rcTextureAlpha** (unsigned char *value*);
If a texture is composed of three elements, the alpha value of a texture is added automatically by *value*. If a texture is composed of four elements, the alpha value of the original texture is used as it is.

5.4 Programming

In order to use textures, all texture data should be loaded in the initialization stage and is called back when needed. Below steps guarantee the proper usage of textures.

[Step 1: Initial setting]

1. Generate a texture object. Identify the name of the texture object.
2. Activate the texture object to be loaded in its name.
3. Load the texture.

[Step 2: Specify if textures will be used when setting the material properties]

1. Activate the texture object to be used in its name.
2. Specify the activation status of the texture.

Texture is considered as a material property and hence must be set in the stage of material property

setting. It can be changed in rendering stage; however, we may expect desirable results if texture is used together with material properties. Therefore, it is recommended that the settings for texture and material properties are made in the initial stage and that the settings remain the same during the rendering process.

5.5 Sample Codes

a. Texture Loading

The explanation of how to load textures is shown below.

1. Create a texture object.
2. Specify by name the texture object
3. Load the texture data

```
rcGenTextures(1, &m_texture[index].bindName);
rcBindTexture(RC_TEXTURE_2D, m_texture[index].bindName);


if(m_texture[index].pImage->type == 3){

      rcTextureAlpha(0x10);
      rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                        m_texture[index].pImage->sizeX,
                        m_texture[index].pImage->sizeY,
                        0, RC_RGB, RC_UNSIGNED_BYTE,
                        m_texture[index].pImage->data);
}
else if(m_texture[index].pImage->type == 4) {

      rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGBA,
                        m_texture[index].pImage->sizeX,
                        m_texture[index].pImage->sizeY,
                        0, RC_RGBA, RC_UNSIGNED_BYTE,
                        m_texture[index].pImage->data);
}
```

Internally, textures use the same form of RGBA. If a texture is set in the form of RGB, an alpha value can be assigned by calling ***rcTextureAlpha***. If the original texture data is in the form of RGBA, it may be used as it is.

b. Specifying the use of textures

The use of textures should be determined in the setting for material properties. Material properties setting will be covered in the next chapter.

```
{
```

```
    rcGenMaterials(1, &sceneData.m_material[i].bindName);
    rcBindMaterial(sceneData.m_material[i].bindName);

    rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, &sceneData.m_material[i].ambient.r);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &sceneData.m_material[i].diffuse.r);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, &sceneData.m_material[i].specular.r);
    rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, sceneData.m_material[i].exponent);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, &sceneData.m_material[i].reflection.r);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_TRANSMITTANCE, &sceneData.m_material[i].refraction.r);
    rcMaterialf( RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, sceneData.m_material[i].refractionIndex);
    rcEnable(RC_TEXTURE_2D);
    rcBindTexture(RC_TEXTURE_2D, sceneData.m_texture[0].bindName);
  }
```

After textures are activated at the configuration stage of material properties, loaded texture objects are specified by their name.

Chapter 6 Material

The material property is a data which is used in rendering and which is a characteristic value that is assigned to a specific object. This value consists of three elements of RGB (red, green and blue). This chapter explains the meaning of each element and its configuration.

6.1 Material Property

Material property of an object can be categorized into (i) a property that is used to determine the color, and (ii) a property that is used to generate the secondary ray. Along with the material property, texture which represents the feel can be specified and be used together with color elements. In order to use the texture, it is applied to the color elements. Reflection, refraction, transmission are the set of elements which determines how to generate the secondary ray. Also, the property that is needed in the secondary ray can be split into three elements and these three elements can be interpreted as the values that will determine the degree of reflection and transmission of RGB.

6.2 Color Property

Color property is an element which is used to perform the light calculation in the process of rendering. This property is classified into and used as three types – ambient, diffuse, and specular.

a. Ambient

Ambient is defined as the light that exists in the surroundings. This property has no specific direction as it illuminates uniformly. The rays generated from the light source gradually fade away and lose direction after they undergo a countless processes of reflection and refraction.

During the daytime, it is possible to see the interior objects clearly even without the light coming into

the house because of the light derived from sunlight is scattered around by refraction and reflection. On the other hand, it is hard to sense the surroundings during the nighttime due to the dissolution of ambient light. This kind of property that is expressed in terms of color property is called ambient.

b. Diffuse

Diffuse is the most common type of light property in 3D graphics. This light property influences the final color of the object due to the incident angle based on the direction of the light source. The degree of lightness and darkness is distinguished based on the location of the light source.

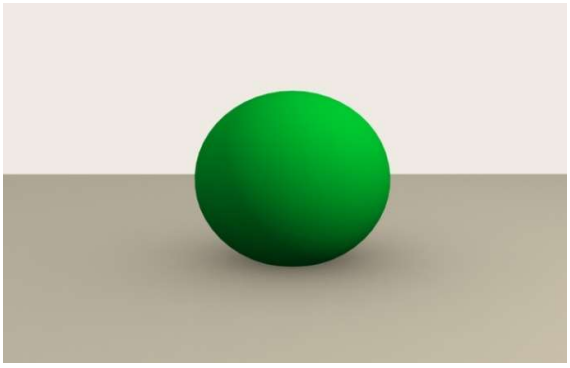

Figure 1. Diffuse

c. Specular

Certain part of the object is shown very brightly when the light is intensively illuminated to make strong reaction on the surface. Such phenomena are expressed using specular. Due to this property, one section of the object is expressed radiantly when the light is situated in a specific location.

The relative brightness in the specific part of the object is made possible due to relatively bright value is acquired from the value of the normal vector of that specific location. Also, it is possible to express highlighted part based on the location of light source.

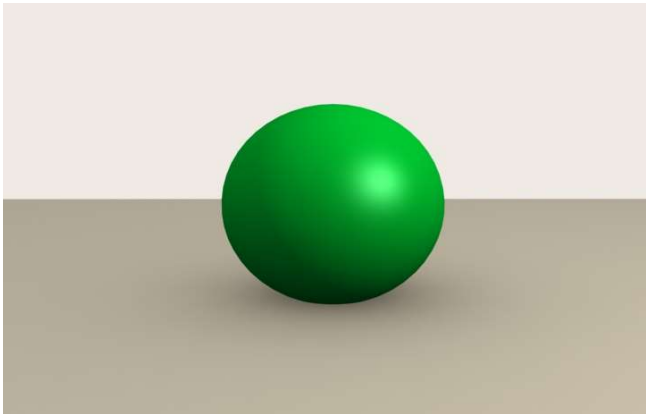

Figure 2. Specular

6.3 Light Property

The properties of light are used to generate the secondary ray. The following essential particular needs special attention when setting the property of the light source.

- The sum of a reflectance and a transmittance should not exceed 1.

As discussed in Chapter 2, if the sum of a reflectance and a transmittance exceeds 1, unintended, unexpected result may occur.

a. Reflection

If the property of reflection is specified to the object, the surrounding objects are reflected on the surface of this object.

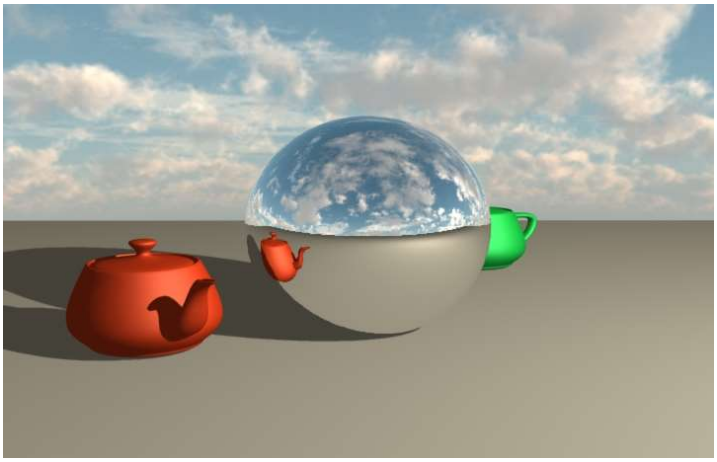

Figure 3. Reflection

b. Transmission

Transmittance is the rate at which the rays penetrate objects. Through this value, the degree of transmitted rays can be specified. Transmittance property sets up the value in form of RGB.

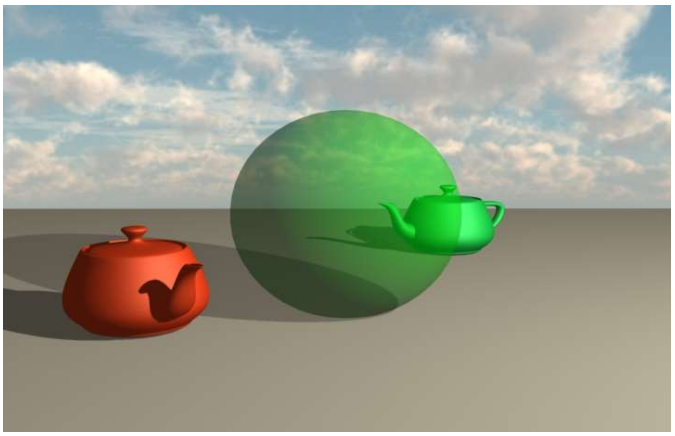

Figure 4. Transmission

c. Refraction

Refractive index is a single value to express the extent to which the rays are refracted on the surface. The direction of a transmitted light is altered based on the refractive index. As shown in Figure 29, it is evident that the expression of overall shadow and kettle shape has been compared to that of Figure 28.

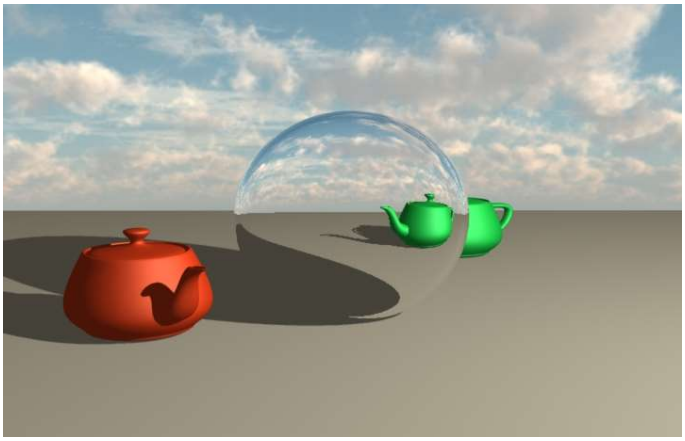

Figure 5. Refraction

6.4 Programming

The number of material objects can be up to '512'. The material object named '0' is set aside to specify a material that will not be used. Therefore, a user is allowed to configure freely using '511' numbers.

OpenGL does not have such limitation regarding the number of material objects; however, it is cumbersome to set up the property value each time. On the other hand, RayCore® 1000 API only requires one-time setup on the material property, allowing users to easily specify the material property of other objects by binding the material object without additional efforts to set up the material property. The only step that is required is the assignment of objects to apply that particular material property. This makes the users to reuse the duplicated material property, thereby simplifying the programming.

The material can be specified using the following functions.

void **rcGenMaterials** (RCsizei *n*, RCuint **materials*);
Generates the material objects. Assigns the number of objects that needs to be created into *n*. When the object is created, the name of the object is received from *materials*. The name starts from the minimum value of '1'. The value '0' is neglected and is only used for the material property initialization purpose.
void **rcBindMaterial** (RCuint *materials*);
Assigns the material object that is to be created. This assigned material object becomes activated. In order to set the material object, it needs to be activated. The activated material object signifies that this will be used in setting material property or in the object which is to be rendered.
void **rcMaterialf** (RCenum *face*, RCenum *pname*, RCfloat *param*); void **rcMaterialfv** (RCenum *face*, RCenum *pname*, const RCfloat **params*);
Is a function which is used to set the material property based in ***rcDraw*** function. *face* only supports **RC_FRONT_AND_BACK**. It is possible to set up the material property using *pname*. In order to assign three color properties, **RC_AMBIENT**, **RC_DIFFUSE**, **RC_SPECULAR**, **RC_REFLECTION**, **RC_TRANSMITTANCE**, **RC_AMBIENT_AND_DIFFUSE** are used. The input data is inserted in a vector format to *params*. **RC_SHINIESS**, **RC_REFRACTION_INDEX** are used in inserting a single value. The input value is assigned in *param*.

- RC_AMBIENT inputs information of three types of color for ambient.
- RC_DIFFUSE inputs information of three types of color for diffuse.
- RC_SPECULAR inputs information of three types of color for specular.
- RC_REFLECTION inputs the information of three colors for reflectance.
- RC_TRANSMITTANCE inputs the information of three colors for transmittance.
- RC_AMBIENT_AND_DIFFUSE simultaneously inputs the color information for ambient and diffuse.
- RC_SHININESS sets the exponent value of specular.
- RC_REFRACTION_INDEX sets the refractive index.

a. Program Model

The material can be specified and be used in the following way:

[Initial setting]
1. Call the name of the material object that is generated.
2. Activate the material object that is to be loaded using its name.
3. Set material properties.

[Rendering]
1. Activate the material object using its name.
2. Load the object which is to be rendered.

b. Notes

The following items must be taken care of when setting and using the material property:

- Remember that the material data is global data
- Remember that the texture is part of the material property configuration

The material object is global data that is used in the overall rendering process. If frequently used as in the case of OpenGL, other rendering object which uses an equivalent material object is rendered by utilizing reconfigured material property. This is made possible by reusing the single material. In order to use the material data while changing its value, the material object should be assigned independently so that the material object is not reused in other object that is to be rendered.

The texture can be activated in the material data configuration. Trying not to set the texture in the rendering process causes less confusion while programming. It is possible to simplify programming through simple configuration of the material object with or without texture usage.

6.5 Code Example

a. Setting the material properties for Phong shading

Assign the basic material properties. Use the material properties which are used in OpenGL. Based on the assigned value, the result will alter accordingly.

```
#include "RCFramework.h"

// Programming Guide - Phong Shading

#include "sphere.h"

class Tutorial : public RCFramework
```

```
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Phong Shading", 800, 480){}
  virtual ~Tutorial(void){}

  RCuint    m_staticMaterial;
  RCuint    m_bindName[4];

protected:
  virtual BOOL OnIntialize(void){

      rcSceneAllInit();

      {
          rcDepthBounce(14);

          rcClearColor(0.5f, 0.5f, 0.5f, 1.0f);
          rcViewport(0, 0, Width(), Height());

          rcMatrixMode(RC_PROJECTION);
          rcLoadIdentity();

          rcuPerspective(30, (float)Width() /(float)Height(), 10, 10000);

          rcuLookAt(0, 0, 8, 0, 0, 0, 0, 1, 0);


            /*
             * Light Setting
             *
             */

          rcEnable(RC_LIGHTING);

          {
              rcEnable(RC_LIGHT0);

              {
                  float pos[4] = {-5, 5, 8, 1};
                  rcLightfv(RC_LIGHT0, RC_POSITION, pos);
              }
              {
                  float ambient[4] = {0, 0, 0, 0};
                  rcLightfv(RC_LIGHT0, RC_AMBIENT, ambient);
              }
              {
                  float diffuse[4] = {0.2f, 0.3f, 0.4f, 0};
                  rcLightfv(RC_LIGHT0, RC_DIFFUSE, diffuse);
              }
              {
                  float specular[4] = {1, 1, 1, 0};
                  rcLightfv(RC_LIGHT0, RC_SPECULAR, specular);
              }
          }


           /*
             * Material Setting
```

```
    *
    */

{
    rcGenMaterials(1, &m_bindName[0]);

    rcBindMaterial(m_bindName[0]);

    {
        float ambient[4] = {0, 0, 0, 0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
    }
    {
        float diffuse[4] = {0, 1, 1, 0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
    }
    {
        float specular[4] = {1, 0, 0, 0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
    }
    {
        float exponent = 10;
        rcMaterialf (RC_FRONT_AND_BACK, RC_SHININESS, exponent);
    }
    rcDisable(RC_TEXTURE_2D);
}

{
    rcGenMaterials(1, &m_bindName[1]);

    rcBindMaterial(m_bindName[1]);

    {
        float ambient[4] = {0, 0, 0, 0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
    }
    {
        float diffuse[4] = {1, 0, 1, 0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
    }
    {
        float specular[4] = {0, 1, 0, 0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
    }
    {
        float exponent = 20;
        rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
    }

    rcDisable(RC_TEXTURE_2D);
}

{
    rcGenMaterials(1, &m_bindName[2]);

    rcBindMaterial(m_bindName[2]);

    {
```

```
            float ambient[4] = {0, 0, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
        }
        {
            float diffuse[4] = {1, 1, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
        }
        {
            float specular[4] = {0, 0, 1, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
        }
        {
            float exponent = 30;
            rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
        }

        rcDisable(RC_TEXTURE_2D);
}

{
        rcGenMaterials(1, &m_bindName[3]);

        rcBindMaterial(m_bindName[3]);
        {
            float ambient[4] = {0, 0, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
        }
        {
            float diffuse[4] = {0, 1, 1, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
        }
        {
            float specular[4] = {0, 0, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
        }
        {
            float exponent = 20;
            rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
        }

        rcDisable(RC_TEXTURE_2D);
}


    /*
     * Static Scene
     *
     */

{

        float galaxyV[12] = {
                -1, 0, -1,
                1, 0, -1,
                1, 0, 1,
                -1, 0, 1
            };
```

```
                float galaxyT[8] = {
                    0, 0,
                    1, 0,
                    1, 1,
                    0, 1
                };

                float diffuse[3] = {0.6f, 0.8f, 0.5f};

                rcGenMaterials(1, &m_staticMaterial);
                rcBindMaterial(m_staticMaterial);

                {
                    float ambient[4] = {0, 0, 0, 0};
                    rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
                }
                {
                    float diffuse[4] = {1, 1, 0, 0};
                    rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
                }
                {
                    float specular[4] = {0, 0, 0, 0};
                    rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
                }
                {
                    float exponent = 10;
                    rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
                }

            rcDisable(RC_TEXTURE_2D);

            rcStaticSceneBegin();

            rcEnableClientState(RC_VERTEX_ARRAY);
            rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

            rcVertexPointer(3, RC_FLOAT, 0, galaxyV);
            rcTexCoordPointer(2, RC_FLOAT, 0, galaxyT);

            rcMatrixMode(RC_MODELVIEW);
            rcLoadIdentity();
            rcTranslatef(0.0, -1, -10);
            rcScalef(8, 1, 10);

            rcDrawArrays(RC_QUADS, 0, 4);

            rcDisableClientState(RC_VERTEX_ARRAY);
            rcDisableClientState(RC_TEXTURE_COORD_ARRAY);

            rcStaticSceneEnd();
               }
        }

        return TRUE;
}

virtual BOOL OnDraw(void){
        static int i = 0;
```

```
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        {
            rcClear(RC_COLOR_BUFFER_BIT | RC_DEPTH_BUFFER_BIT);

            rcMatrixMode(RC_MODELVIEW);
            rcLoadIdentity();
            rcTranslatef(0.0, 0, -7.0);
            rcEnableClientState(RC_VERTEX_ARRAY);
            rcEnableClientState(RC_NORMAL_ARRAY);

            rcVertexPointer(3, RC_FLOAT, sizeof(struct Pos), g_SphereVertices);
            rcNormalPointer(RC_FLOAT, 0, g_SphereNormals);

            rcPushMatrix();
            rcBindMaterial(m_bindName[0]);
            rcTranslatef(2, 0, 0);
            rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_SphereVertices)/sizeof(Pos));
            rcPopMatrix();

            rcPushMatrix();
            rcBindMaterial(m_bindName[1]);
            rcTranslatef(-2, 0, 0);
            rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_SphereVertices)/sizeof(Pos));
            rcPopMatrix();

            rcBindMaterial(m_bindName[2]);
            rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_SphereVertices)/sizeof(Pos));

            rcDisableClientState(RC_VERTEX_ARRAY);
            rcDisableClientState(RC_NORMAL_ARRAY);
        }

        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

The material properties are set differently on each object and based on these results, different color results can be earned as follows.

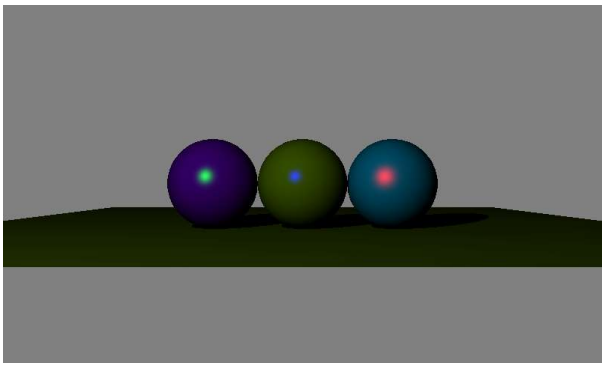

Figure 6. Colored balls by changing the material properties

b. Reflection

Following is an example of a mirror-like reflection. Among two different rectangles, the large one is a frame, and the small one is a mirror. Thus, by setting the reflection property of the material on the small rectangle, the results of reflection on the triangle which is located in front of the small rectangle can be achieved.

```
#include "RCFramework.h"

// Programming Guide - Reflection

typedef struct RGBImageRec {
  int sizeX, sizeY;
  unsigned char *data;
} RGBImageRec;

typedef struct bmpBITMAPFILEHEADER{
  WORD    bfType;
  DWORD  bfSize;
  WORD    bfReserved1;
  WORD    bfReserved2;
  DWORD  bfOffBits;
} BITMAPFILEHEADER1;

typedef struct bmpBITMAPINFOHEADER{
  DWORD  biSize;
  DWORD  biWidth;
  DWORD  biHeight;
  WORD    biPlanes;
  WORD    biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  DWORD  biXPelsPerMeter;
  DWORD  biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPFILEHEADER2;

RGBImageRec *DIBImageLoad(char* path, int channel) {
```

```
RGBImageRec* pImage=NULL;
FILE *f=NULL;
unsigned char *pBuf=NULL;
int dataSize=0;
int index=0;
DWORD x=0;
DWORD y=0;
int bpp=0;

if(channel !=3 && channel !=4) return pImage;

f = fopen(path, "rb");
if(f != NULL) {
	BITMAPFILEHEADER1 HD1;
	BITMAPFILEHEADER2 HD2;

	fseek(f, 0, SEEK_SET);
	fread(&HD1.bfType, sizeof(WORD), 1, f);
	fread(&HD1.bfSize, sizeof(WORD), 1, f);
	fseek(f, 10, SEEK_SET);
	fread(&HD1.bfOffBits, sizeof(int),1, f);

	fread(&HD2.biSize, sizeof(int), 1, f);
	fread(&HD2.biWidth, sizeof(int), 1, f);
	fread(&HD2.biHeight, sizeof(int), 1, f);
	fread(&HD2.biPlanes, sizeof(WORD), 1, f);
	fread(&HD2.biBitCount, sizeof(WORD), 1, f);
	fread(&HD2.biCompression, sizeof(int), 1, f);

	fseek(f,HD1.bfOffBits,SEEK_SET);

	bpp = HD2.biBitCount/8;
	if(bpp == 1 || bpp == channel)
	{
		pBuf = (unsigned char*) malloc(channel);

		pImage = (RGBImageRec*) malloc(sizeof(RGBImageRec));
		pImage->sizeX = HD2.biWidth;
		pImage->sizeY = HD2.biHeight;

		dataSize = HD2.biWidth*HD2.biHeight*channel;
		pImage->data = (unsigned char*) malloc(dataSize);

		for(y=0; y<HD2.biHeight; y++) {
			for(x=0; x<HD2.biWidth; x++) {
				fread(pBuf, bpp, 1, f);
				if(bpp == 1) {
					pBuf[1] = pBuf[2] = pBuf[0];
					if(channel == 4)	pBuf[3] = 0;
				}

				index = (y*HD2.biWidth + x)*channel;
				pImage->data[index]	= pBuf[2];
				pImage->data[index + 1]  = pBuf[1];
				pImage->data[index + 2]  = pBuf[0];
				if(channel == 4)
					pImage->data[index + 3] = pBuf[3];
			}
```

```
            }

            if(pBuf) free(pBuf);
            pBuf = NULL;
        }

        fclose(f);
    }

    return pImage;
}

RCfloat         m_Angle=0.0f;
RGBImageRec     *g_texture;
RCuint          *g_textureName;

#define MATERIAL_FRAME          0
#define MATERIAL_MIRROR         1
#define MATERIAL_TRIANGLE       2
unsigned int g_MaterialArray[3];

class Tutorial : public RCFramework
{
public:
    Tutorial(void) : RCFramework("Programming Guide - Reflection", 800, 480){}
    virtual ~Tutorial(void){}

    void GenTexture(void){
        g_texture = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Reflection/triangle.bmp", 3);

        if(g_texture){
            g_textureName = new RCuint[1];

            rcGenTextures(1, g_textureName);
            rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);
            rcTextureAlpha(128);
            rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                         g_texture[0].sizeX, g_texture[0].sizeY, 0,
                         RC_RGB, RC_UNSIGNED_BYTE,
                         g_texture[0].data);

            if(g_texture->data)   free(g_texture->data);
            free(g_texture);
        }
    }

    void GenMaterial(void){
        RCfloat Reflectance[] = {1.0, 1.0, 1.0};
        RCfloat NotReflectance[] = {0, 0, 0};
        RCfloat color[3][4]={
            {0.2f, 0.4f, 0.8f, 0.0f},
            {0.0f, 0.8f, 0.4f, 0.0f},
            {0.0f, 1.0f, 0.0f, 0.0f},
        };

        rcGenMaterials(3, g_MaterialArray);

        rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);
```

```
	rcDisable(RC_TEXTURE_2D);
	rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, &color[MATERIAL_FRAME][0]);
	rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);

	rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);
	rcDisable(RC_TEXTURE_2D);
	rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, &color[MATERIAL_MIRROR][0]);
	rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, Reflectance);

	rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
	rcDisable(RC_TEXTURE_2D);
	rcMaterialfv(RC_FRONT_AND_BACK,
			RC_AMBIENT_AND_DIFFUSE,
			&color[MATERIAL_TRIANGLE][0]);

	RCfloat ambient[] = {0.8,0.8,0.8,0};
	RCfloat diffuse[] = {0.2,0.2,0,0};
	rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
	rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);

	rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);
}

void RenderMirror(void){
	RCfloat QuadVertices[4][3];

	QuadVertices[0][0]=2.0f;
	QuadVertices[0][1]=-2.0f;
	QuadVertices[0][2]=0.0f;

	QuadVertices[1][0]=2.0f;
	QuadVertices[1][1]=2.0f;
	QuadVertices[1][2]=0.0f;

	QuadVertices[2][0]=-2.0f;
	QuadVertices[2][1]=2.0f;
	QuadVertices[2][2]=0.0f;

	QuadVertices[3][0]=-2.0f;
	QuadVertices[3][1]=-2.0f;
	QuadVertices[3][2]=0.0f;

	rcDisable(RC_TEXTURE_2D);

	rcEnableClientState(RC_VERTEX_ARRAY);

	rcMatrixMode(RC_MODELVIEW);
	rcLoadIdentity();

	rcVertexPointer(3, RC_FLOAT, 0, QuadVertices);

	rcPushMatrix();
	rcTranslatef(0, 0, -10);
	rcScalef(2, 2, 2);
	rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);
	rcDrawArrays(RC_QUADS,0,4);
	rcPopMatrix();
```

```
        //mirror
        rcPushMatrix();
        rcTranslatef(0, 0, -9.999f);
        rcScalef(1.7f, 1.7f, 1.7f);
        rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);
        rcDrawArrays(RC_QUADS, 0, 4);
        rcPopMatrix();

        rcDisableClientState(RC_VERTEX_ARRAY);
}

void RenderTriangle(void){
        RCfloat TriangleVertices[3][3];

        TriangleVertices[0][0] = 0.0f;
        TriangleVertices[0][1] = 0.707f;
        TriangleVertices[0][2] = 0.0f;

        TriangleVertices[1][0] = -1.0f;
        TriangleVertices[1][1] = -0.707f;
        TriangleVertices[1][2] = 0.0f;

        TriangleVertices[2][0] = 1.0f;
        TriangleVertices[2][1] = -0.707f;
        TriangleVertices[2][2] = 0.0f;

        rcEnableClientState(RC_VERTEX_ARRAY);

        rcVertexPointer(3, RC_FLOAT, 0, TriangleVertices);

        rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

        float TexIndex[3][2];
        TexIndex[0][0] = 0.5f;      TexIndex[0][1] = 0.f;
        TexIndex[1][0] = 0.f;       TexIndex[1][1] = 1.f;
        TexIndex[2][0] = 1.f;       TexIndex[2][1] = 1.f;

        rcTexCoordPointer(2, RC_FLOAT, 0, TexIndex);

        rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
        rcEnable(RC_TEXTURE_2D);
        rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcPushMatrix();
        rcTranslatef(0, 0, -5);
        rcRotatef(m_Angle, 0, 1, 0);
        rcDrawArrays(RC_TRIANGLES, 0, 3);
        rcPopMatrix();

        rcDisableClientState(RC_TEXTURE_COORD_ARRAY);
        rcBindMaterial(0);
        rcBindTexture(RC_TEXTURE_2D, 0);
        rcDisable(RC_TEXTURE_2D);

        rcDisableClientState(RC_VERTEX_ARRAY);
```

```
    }

    void StaticSceneDraw(void){
        rcStaticSceneBegin();
        RenderMirror();
        rcStaticSceneEnd();
    }

    void DynamicSceneDraw(void){
        RenderTriangle();
    }

protected:
    virtual BOOL OnIntialize(void){
        GenTexture();
        GenMaterial();

        rcSceneAllInit();

        {
            rcClearColor(0.2f, 0.2f, 0.2f, 1.0f);
            rcViewport(0, 0, Width(), Height());

            rcMatrixMode(RC_PROJECTION);
            rcLoadIdentity();

            rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 1.0f, 500.0f);
            rcuLookAt(-3, 0, 0, 0, 0, -8, 0, 1, 0);

            StaticSceneDraw();
        }
        return TRUE;
    }

    virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        m_Angle += 10.f;

        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

As the triangle rotates, the reflected image is seen on the rectangular mirror.

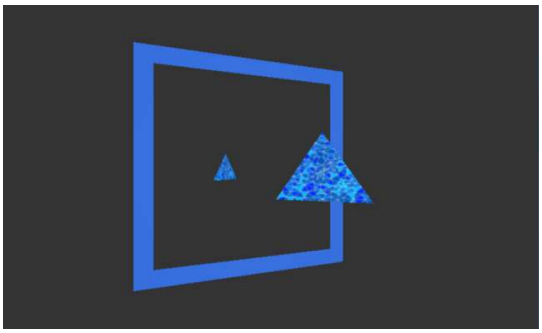

Figure 7. Reflection

c. Transmission and Refraction

Assign the transmittance and refractive index of the material to the rectangle which is located in front of a triangle. Through the triangle, it is possible to achieve transmitted and refracted results.

```
#include "RCFramework.h"

// Programming Guide - Refraction

typedef struct RGBImageRec {
  int sizeX, sizeY;
  unsigned char *data;
} RGBImageRec;

typedef struct bmpBITMAPFILEHEADER{
  WORD    bfType;
  DWORD  bfSize;
  WORD    bfReserved1;
  WORD    bfReserved2;
  DWORD  bfOffBits;
} BITMAPFILEHEADER1;

typedef struct bmpBITMAPINFOHEADER{
  DWORD  biSize;
  DWORD  biWidth;
  DWORD  biHeight;
  WORD    biPlanes;
  WORD    biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  DWORD  biXPelsPerMeter;
  DWORD  biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPFILEHEADER2;

RGBImageRec *DIBImageLoad(char* path, int channel) {
  RGBImageRec* pImage=NULL;
  FILE *f=NULL;
  unsigned char *pBuf=NULL;
  int dataSize=0;
```

```
int index=0;
DWORD x=0;
DWORD y=0;
int bpp=0;

if(channel !=3 && channel !=4) return pImage;

f = fopen(path, "rb");
if(f != NULL) {
	BITMAPFILEHEADER1 HD1;
	BITMAPFILEHEADER2 HD2;

	fseek(f, 0, SEEK_SET);
	fread(&HD1.bfType, sizeof(WORD), 1, f);
	fread(&HD1.bfSize, sizeof(WORD), 1, f);
	fseek(f, 10, SEEK_SET);
	fread(&HD1.bfOffBits, sizeof(int),1, f);

	fread(&HD2.biSize, sizeof(int), 1, f);
	fread(&HD2.biWidth, sizeof(int), 1, f);
	fread(&HD2.biHeight, sizeof(int), 1, f);
	fread(&HD2.biPlanes, sizeof(WORD), 1, f);
	fread(&HD2.biBitCount, sizeof(WORD), 1, f);
	fread(&HD2.biCompression, sizeof(int), 1, f);

	fseek(f,HD1.bfOffBits, SEEK_SET);

	bpp = HD2.biBitCount/8;
	if(bpp == 1 || bpp == channel)
	{
		pBuf = (unsigned char*) malloc(channel);

		pImage = (RGBImageRec*) malloc(sizeof(RGBImageRec));
		pImage->sizeX = HD2.biWidth;
		pImage->sizeY = HD2.biHeight;

		dataSize = HD2.biWidth*HD2.biHeight*channel;
		pImage->data = (unsigned char*) malloc(dataSize);

		for(y=0; y<HD2.biHeight; y++) {
			for(x=0; x<HD2.biWidth; x++) {
				fread(pBuf, bpp, 1, f);
				if(bpp == 1) {
					pBuf[1] = pBuf[2] = pBuf[0];
					if(channel == 4)	pBuf[3] = 0;
				}

				index = (y*HD2.biWidth + x)*channel;
				pImage->data[index]	= pBuf[2];
				pImage->data[index + 1]  = pBuf[1];
				pImage->data[index + 2]  = pBuf[0];
				if(channel == 4)
					pImage->data[index + 3] = pBuf[3];
			}
		}

		if(pBuf) free(pBuf);
		pBuf = NULL;
```

```
        }

        fclose(f);
    }

    return pImage;
}

float g_refractionIndex1 = 1.0f;
float g_refractionIndex2 = 1.0f;

RGBImageRec    *g_texture;
RCuint         *g_textureName;

#define MATERIAL_TRIANGLE       0
#define MATERIAL_REFRACTION1    1
#define MATERIAL_REFRACTION2    2
unsigned int g_MaterialArray[3];

class Tutorial : public RCFramework
{
public:
    Tutorial(void) : RCFramework("Programming Guide - Refraction", 800, 480){}
    virtual ~Tutorial(void){}

    void GenTexture(void){
        g_texture = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Refraction/texture0.bmp", 3);

        if(g_texture){
            g_textureName = new RCuint[1];

            rcGenTextures(1, g_textureName);
            rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);
            rcTextureAlpha(128);
            rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                         g_texture[0].sizeX, g_texture[0].sizeY, 0,
                         RC_RGB, RC_UNSIGNED_BYTE,
                         g_texture[0].data);

            if(g_texture->data)   free(g_texture->data);
            free(g_texture);
        }
    }

    void GenMaterial(void){
        float transmittance[3] = {0.5, 0.5, 0.5};
        RCfloat color[3][4]={
            {0.8f, 0.3f, 0.4f, 0.0f},
            {0.2f, 0.4f, 0.8f, 0.0f},
            {0.4f, 0.8f, 0.2f, 0.0f},
        };

        rcGenMaterials(3, g_MaterialArray);

        rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK,
                     RC_AMBIENT_AND_DIFFUSE,
```

```
                    &color[MATERIAL_TRIANGLE][0]);

    rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION1]);
    rcDisable(RC_TEXTURE_2D);
    rcMaterialfv(RC_FRONT_AND_BACK,
                    RC_AMBIENT_AND_DIFFUSE,
                    &color[MATERIAL_REFRACTION1][0]);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_TRANSMITTANCE, transmittance);
    rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex1);

    rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION2]);
    rcDisable(RC_TEXTURE_2D);
    rcMaterialfv(RC_FRONT_AND_BACK,
                    RC_AMBIENT_AND_DIFFUSE,
                    &color[MATERIAL_REFRACTION2][0]);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_TRANSMITTANCE, transmittance);
    rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex2);
}

void RenderTriangle(void){
    RCfloat TriangleVertices[3][3];

    TriangleVertices[0][0] = 0.0f;
    TriangleVertices[0][1] = 0.707f;
    TriangleVertices[0][2] = 0.0f;

    TriangleVertices[1][0] = -1.0f;
    TriangleVertices[1][1] = -0.707f;
    TriangleVertices[1][2] = 0.0f;

    TriangleVertices[2][0] = 1.0f;
    TriangleVertices[2][1] = -0.707f;
    TriangleVertices[2][2] = 0.0f;

    rcEnableClientState(RC_VERTEX_ARRAY);

    rcVertexPointer(3, RC_FLOAT, 0, TriangleVertices);

    rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

    float TexIndex[3][2];
    TexIndex[0][0] = 0.5f;      TexIndex[0][1] = 0.f;
    TexIndex[1][0] = 0.f;       TexIndex[1][1] = 1.f;
    TexIndex[2][0] = 1.f;       TexIndex[2][1] = 1.f;
    rcTexCoordPointer(2, RC_FLOAT, 0, TexIndex);

    rcMatrixMode(RC_MODELVIEW);
    rcLoadIdentity();

    rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
    rcEnable(RC_TEXTURE_2D);
    rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);

    rcPushMatrix();
    rcTranslatef(0, 0, -10);
    rcScalef(3, 3, 3);
    rcDrawArrays(RC_TRIANGLES, 0, 3);
    rcPopMatrix();
```

```
    rcDisableClientState(RC_TEXTURE_COORD_ARRAY);
    rcBindMaterial(0);
    rcBindTexture(RC_TEXTURE_2D, 0);
    rcDisable(RC_TEXTURE_2D);

    rcDisableClientState(RC_VERTEX_ARRAY);
}

void RenderRefraction(void){
    RCfloat QuadVertices[4][3];

    QuadVertices[0][0] = 2.0f;
    QuadVertices[0][1] = -2.0f;
    QuadVertices[0][2] = 0.0f;

    QuadVertices[1][0] = 2.0f;
    QuadVertices[1][1] = 2.0f;
    QuadVertices[1][2] = 0.0f;

    QuadVertices[2][0] = -2.0f;
    QuadVertices[2][1] = 2.0f;
    QuadVertices[2][2] = 0.0f;

    QuadVertices[3][0] = -2.0f;
    QuadVertices[3][1] = -2.0f;
    QuadVertices[3][2] = 0.0f;

    rcEnableClientState(RC_VERTEX_ARRAY);

    rcVertexPointer(3, RC_FLOAT, 0, QuadVertices);

    rcMatrixMode(RC_MODELVIEW);
    rcLoadIdentity();

    rcPushMatrix();
    rcTranslatef(0, 0, -5);

    rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION1]);
    rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex1);
    rcDrawArrays(RC_QUADS, 0, 4);

    rcPopMatrix();

    rcDisableClientState(RC_VERTEX_ARRAY);
}

void RefreshRefraction(){
    rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION1]);
    rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex1);
}

void StaticSceneDraw(void){
    rcStaticSceneBegin();
    RenderTriangle();
    RenderRefraction();
    rcStaticSceneEnd();
}
```

```
  void DynamicSceneDraw(void){
        RefreshRefraction();
  }

protected:
  virtual BOOL OnIntialize(void){
        GenTexture();
        GenMaterial();

        rcSceneAllInit();

        {
              rcClearColor(0.4f, 0.3f, 0.2f, 0.0f);
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();

              rcuPerspective(70.0f, (RCfloat)Width() / (RCfloat)Height(), 1.0f, 500.0f);
              rcuLookAt(-3, 0, 0, 0, 0, -8, 0, 1, 0);

              StaticSceneDraw();
        }
        return TRUE;
  }

  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        g_refractionIndex1 += 0.1f;
        g_refractionIndex2 -= 0.1f;

        return TRUE;
  }
};

Tutorial      g_Tutorial;
```

As the refractive index changes, the transmitted and refracted results can be shown accordingly.

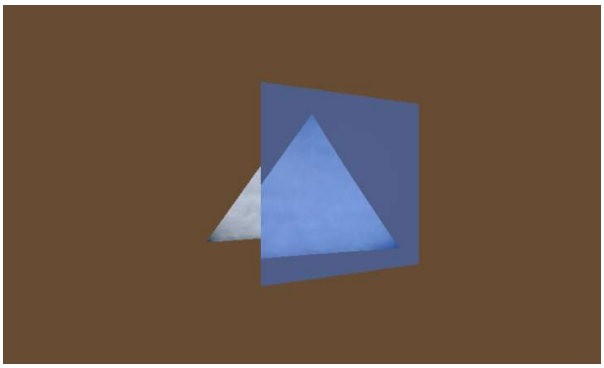

Figure 8. Transmission and refraction

Chapter 7 Drawing

This chapter explains how an object is sent to the renderer and expressed as an image. An object is a set of virtual triangles with their own material properties. This set of triangles can be expressed in various forms of shapes, which exist in a three dimensional space where the transformation of coordinates is performed.

The essential reason for drawing an object is to calculate the final color of a pixel by understanding the optical correlation properties. This arithmetic process is called shading. The most popular algorithm in real-time rendering is Phong shading. Also known as Phong illumination model, this algorithm works with various properties such as a light's incidence and reflection angle, and an object's ambient, diffuse and specular light. These material properties may be characterized separately during the rendering process in order to express various light effects.

In rendering, the triangles are created based on the information of their actual coordinates, which add up to constructing objects given with their own material properties. As mentioned earlier, an object consists of a set of triangles or primitives, one of which is the minimum unit in a three dimensional space. The coordinates of triangles eventually determine the shape of an object in a three dimensional space.

7.1 Triangle

Triangles play a very important role in 3D graphics, one of which is the smallest unit used to represent an object. Points, lines and surfaces also play the similar role; however, there are some technical difficulties to express lines and point in a three dimensional space. Theoretically, the sum of points form a line, and the sum of lines form a surface. When it gets closer to a line with the significant number of points, there may exist some spaces in between the points. In this case, the number of points may not be sufficient. The challenge arises from the fact that it is hard to decide

how many points are sufficient enough to express a line, since the unlimited number of midpoints may exist in between two points forming a line. Using segments may also be considered as an alternative way to construct a line. That is, the midpoints in between two points can be created repeatedly in every arithmetic operation, the number of the midpoints created will be sufficient enough. Therefore, a surface can be expressed in the same way. This process is called the vector graphics.

A surface can be formed using multiple triangles. In addition, a curved surface such as the surface of a ball can also be expressed by a set of smaller triangles. Triangles are used for creating not only for flat surfaces but also complicated objects. This is why a triangle is used as a basic unit.

a. Vertex List

In a triangle configuration, vertex data is collected in a set. One triangle consists of three vertices. In order to generate a triangle, two methods are used. The first method generates a triangle by storing three vertices sequentially. The second method uses the index values of the stored vertices.

In general, the data of a triangle's vertices has at least three coordinate values and their corresponding information such as normal vectors and texture coordinates. On the other hand, the indices consist of integers. In the first case, the data size may increase if a triangle is made by suing the same coordinates more than once. However, in the second case, the actual data size of vertices may decrease even with the overlap. Even though the memory usage may decrease with the smaller data size of vertices, additional index information may be needed as the data size becomes smaller. Therefore, the optimal method should be used selectively depending on the types of storing objects.

b. Vertex Data

Vertex data can be stored in various structures. Necessary coordinates in (x, y, z) must be stored in the same type in the same order of x, y and z correspondingly. Vertex information can be stored together with the information of material properties. In this case, whether to store vertex information with the information of material properties need to be specified by a function that transfers the data.

void **rcVertexPointer** (RCint *size*, RCenum *type*, RCsizei *stride*, const RCvoid *pointer);
This function is used to input the vertex data stored in an array. It provides the *size* and *type* information of the input data for vertices. *size*, the number of coordinates(x, y, z) for vertex data, is always 3. *type* supports **RC_BYTE**, **RC_SHORT**, **RC_FLOAT** and **RC_FIXED**. *stride* specifies the byte offset between vertices. *stride* is either 0 or the actual size if the data consists of only the coordinates of each vertex. This function inputs the address of the actual data in *pointer*.

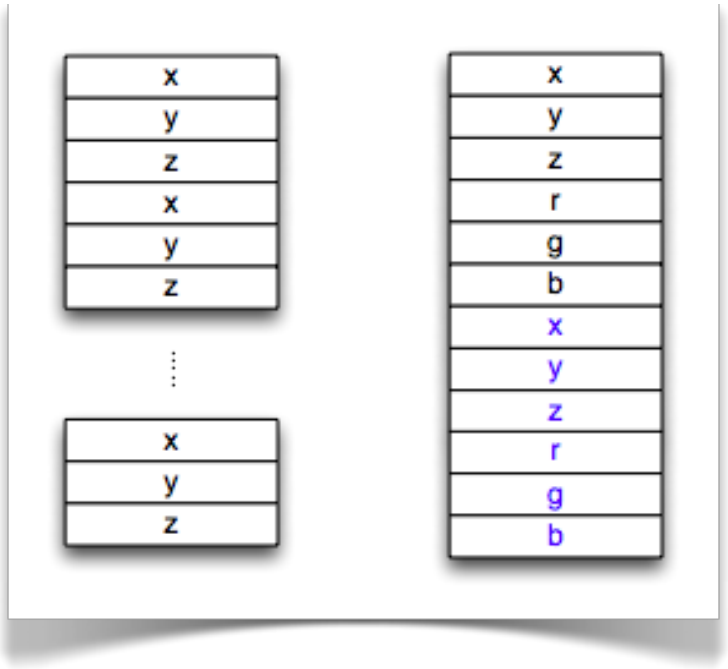

Figure 31. Data storing

void **rcTexCoordPointer** (RCint *size*, RCenum *type*, RCsizei *stride*, const RCvoid *pointer);
This function inputs the array of texture coordinates data for vertices. It also provides the *size* and *type* information of the input data for vertices. *size*, the number of texture coordinates (u, v) is only 2. *type* supports **RC_BYTE**, **RC_SHORT**, **RC_FLOAT** and **RC_FIXED**. *stride* specifies the byte offset between the texture coordinates of two adjacent vertices. If the data consists of only the texture coordinates of each vertex, 0 or the actual value can be used. This function inputs the address of the actual data in *pointer*.

Since only 2D textures are supported, only 2 texture coordinates in (u, v) should be used.

void **rcNormalPointer** (RCenum *type*, RCsizei *stride*, const RCvoid *pointer);
This function inputs the array of data for normal vectors. Also, it tells the *type* information of the data. *type* supports **RC_BYTE**, **RC_SHORT**, **RC_FLOAT** and **RC_FIXED**. *stride* specifies the byte offset between normal vectors. If the data consists of only the coordinates of each normal vector, 0 or the actual size is used. This function inputs the address of the actual data in *pointer*.

Since normal vectors are always expressed by three coordinates in (x, y, z), *size* is not assigned. When a vertex array is configured, the position and the material properties of an object should not be included in a single array. For example, (XRYGZB) does not exist. The location information as well as other types of information for an object should be stored sequentially.

Each component consists of three elements. The information of location is expressed by XYZ, whereas the information of color is expressed by RGB. However, the information of texture coordinates is expressed by two elements, UV. Other cases are not supported in the current version of RayCore® 1000 API.

7.2 Creation of Triangles

Vertex data is input using vertex arrays. If the data is composed of triangles in a sequential order, it can be used as it is. Otherwise, vertex indices should be applied to compose a new triangle. When vertex indices are implemented, no vertices need to be duplicated when their data is stored; therefore, the data size can be reduced. Depending on the situation, using vertex indices becomes a very useful method in the aspect of data size management. RayCore® 1000 API supports both methods above.

a. Vertex Sequence

Since vertex data is used repeatedly, the efficiency of memory usage decreases. However, sequential processing is possible.

void **rcDrawArray** (RCenum *mode*, RCint *first*, RCsizei *count*);
This function generates triangles based on the order of each vertex being stored. *mode* illustrates the method of creating triangles, which supports **RC_TRIANGLES**, **RC_QUADS**, **RC_TRIANGLE_FAN** and **RC_TRIANGLE_STRIP**. Calling ***rcVertexPointer*** determines *first*, the index for the first vertex of the first triangle in an array of vertices. *count* specifies the number of vertices to be processed. This last index becomes *first+count-1*.

b. Vertex Index List

Since identical vertex data is shared, memory usage significantly decreases. However, the bandwidth may become bigger due to the non-sequential order of vertices.

void **rcDrawElements** (RCenum *mode*, RCsizei *count*, RCenum *type*, RCvoid ***indices*);
This function generates triangles based on the order of each vertex being stored. *mode* illustrates the method of creating triangles, which supports **RC_TRIANGLES**, **RC_QUADS**, **RC_TRIANGLE_FAN** and **RC_TRIANGLE_STRIP**. *count* specifies the number of indices in an index list for the array determined by ***rcVertexPointer***. *type* specifies the data type of the list. *indices* specifies the data address of the list.

Chapter 8 Rendering Start

Because RayCore® 1000 uses backward rendering, the starting point of rendering should be specified. After all the objects to render are loaded, the rendering process commences.

Below function specifies the starting point of rendering.

void **rcFinish** (void);
This function indicates the starting point of rendering. It also determines if the target data is loaded completely before staring rendering.

8.1 Example

rcFinish() has no sample codes because it is already specified inside ***RCFrameWork*** explained in Appendix A. Therefore, it does not need to be called separately.

Chapter 9 Static/Dynamic Objects

After loading all objects into memory, which are necessary to generate images, RayCore® 1000 starts rendering. Also, the acceleration structure of each data is created before RayCore® 1000 renders the objects by ray tracing. The generation of acceleration structures requires repetitive computational operations which lead to the decrease in the overall rendering speed. This chapter introduces the concept of static and dynamic objects which resolve the problem of low rendering speed with many computations for creating acceleration structures.

The most typical example of application programs in real-time 3D graphics is games where both static and dynamic objects are widely applied. Static objects mostly express backgrounds in games, whereas dynamic objects are moving within the backgrounds. Musicals may demonstrate an excellent example of the application of static and dynamic objects. In musicals, stages may be considered as static objects, which do not change until certain act is finished. However, actors keep moving around the stage within the current single act and therefore may represent dynamic objects.

There is no such distinction in rasterization, because all triangles or primitives are processed in a sequential order. However, since primitives are recycled in ray tracing, the stored primitives need to be maintained continuously. Similarly, if there are no changes made to static objects, the exact information of the static objects stored in memory can be recycled. As a result, rendering speed can be improved by decreasing the significant amount of costs for creating acceleration structures and loading data.

When data is distinguished between static and dynamic objects, the speed of data processing can be improved. Below explains how an object is given static property.

9.1 Object Classification

An object can be expressed with three-dimensional coordinates. An object consists of a single triangle or multiple triangles. For example, multiple triangles form an object which may express an arm or a leg of an animation character, or form a map in a space where characters are moving. Such classification of objects can be decided according to the scene or can be decided according to the specified scenario which belongs to the scene.

In rasterization, all objects are regarded as only dynamic objects. RayCore® 1000 begins to generate images after the transfer of all data to the renderer is completed. Here, RayCore® 1000 renderer does not distinguish objects based on the movement status, but based on whether or not to render the objects. In ray tracing, however, it is not easy to decide whether to render an object, since objects outside frustum may affect final images by the reflection and refraction of rays. Basically, the renderer is not capable of identifying an object is static or moving. The renderer is purely responsible for generating images based on the input data. In other words, it does not have a capability to interpret images. Classifying the dynamic and static objects is made depending on whether the renderer recreates the acceleration structure.

The dynamic or static status of an object needs to be managed separately by users in application programs.

a. Dynamic Object

Dynamic objects are objects that are moving in a certain scene. If a stationary object starts moving, it becomes a dynamic object. In general, a dynamic object means a moving object. In application programs such as games, however, dynamic objects may refer to anything that may move.

Designating an object as a dynamic object means the repetitive creation of acceleration structures. In general, building acceleration structures is considered as a process that is performed prior to rendering. In ray tracing or interactive rendering, however, it should be understood as a process performed during the rendering process.

An object that is not static is identified as a dynamic object. In other words, every input data has potential dynamic status.

In games, characters are good examples of dynamic objects. Characters that are continuously moving have interactive relationships with users. Objects such as doors and windows mostly don’t move but can still be designated as dynamic objects, because doors can be opened or closed in a given situation. Of course, characters that do not need to move at all may be given the static status.

b. Static Object

An object is given the static status by a programmer when it is not expected to move or does not need to move in a certain scene. When an object becomes a static object, the object is assumed not moving.

When an object is specified as a static object, the acceleration structure for that object is recycled without having to be recreated for every scene. This recycling reduces the computations for building acceleration structures. Modification of acceleration structures that are already created is not recommended, since it may deteriorate rendering performance. Hence, acceleration structures should be regenerated if they need modification for some reason. In this case, the repetition of creating acceleration structures makes no difference with using dynamic objects. In other words, if the objects do not need acceleration structures for a certain period of time, setting those objects as static objects would be more efficient. For instance, the backgrounds and landscapes in games, which do not need to be updated for a given scene, may be given the static property.

c. Static Objects with Dynamic Property

Imagine an example in which some part of a static object is moving. In this case, the acceleration structures for the other part of the static object as well as other dynamic objects should be recreated. This case makes no difference with originally setting that object as a dynamic object.

When a static object is changed to a dynamic object, it becomes ambiguous to determine if the object is static or dynamic. When the movement status of objects changes, distinguishing the objects as dynamic or static objects needs to be determined efficiently. In some cases, the composition form of the static and dynamic objects may affect rendering speed. In this case, the separate management for those objects needs to be implemented and hence may cause additional computational costs.

RayCore® 1000 API does not support automatic processing for objects with both properties, because the processing type of upper-level application programs are different from each other. Therefore, a separate management for such type of objects needs to be considered for each application program.

Important key points are the following:

- Acceleration structures for static objects will not be recreated unless they are initialized.
- When static objects are not used, object information needs to be initialized so that their data is not used.
- Objects that are not selected as static objects are considered as dynamic objects.

The explanation of how to give the static property to objects will be followed.

9.2 Static Object Programming

For dynamic objects, the process explained above can be applied whenever a new scene is generated. If objects are changed depending on the input value, new data should be loaded after the coordinates of the objects are transformed to the new position.

a. Programming Model

Simple begin-end model can be used to specify static objects. It uses “begin” function that signals the beginning of a static object and “end” function that signals the end of the static object.

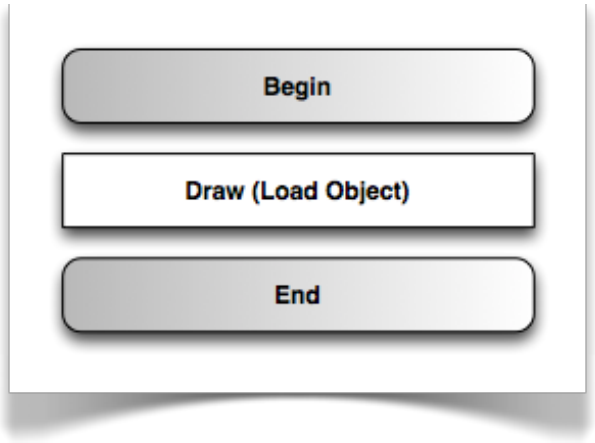

Figure 32. Beginning and end of a static object

b. Function List

When specifying a static object, the functions for basic loading, transformation, and material setting are used in the same way as the functions are used for dynamic objects. The beginning and end of loading a static object are also specified. RayCore® 1000 API supports the function for such process. In addition, RayCore® 1000 API supports a separate function uses or deletes static objects.

Below is the function that initializes or deletes static objects.

void **rcSceneAllInit** (void);
This function initializes static objects. When static objects are being used, this function deletes these static objects.

Below is the function that indicates the beginning and end of loading static objects.

void **rcStaticSceneBegin** (void);
This function tells when loading static object data begins. All data loaded without the function called is stored as data for dynamic objects. Therefore, when static objects are needed, this function must be called.

Below is the function that indicates the completion of loading static objects.

void **rcStaticSceneEnd** (void);
This function indicates the completion of loading static objects. It also tells when the creation of acceleration structures begins.

9.3 Notes

There are a couple of programming rules that need to be satisfied for static objects.

- The static object should not be specified separately.
- The static object data should be loaded in front of memory area for rendering data.

Based on the above programming rules, users need to keep in mind the points below.

- Only single static object is used in a single scene.
- Static objects need to be created in prior to dynamic objects.

9.4 Sample Codes (Cornell Box*)

The following examples show how to define static background by using extended functions.

a. Initialization

Basic initial parameters are defined as follows:

```
rcSceneAllInit();

rcClearColor(0.0f, 0.2f, 0.4f, 1.0f);

rcMatrixMode(RC_PROJECTION);
rcLoadIdentity();
```

* Designed in Cornell University

```
rcFrustum(-1, 1, -0.6f, 0.6f, 1.7, 1000);

rcuLookAt(27.8, 27.3, -80.0, 27.8, 27.3, 0, 0, 1, 0);

rcStaticSceneBegin();
rcEnableClientState(RC_VERTEX_ARRAY);

rcVertexPointer(3, RC_FLOAT, 0, Left_Wall);
rcGenMaterials(1, &materialID[0]);
rcBindMaterial(materialID[0]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE,&Red.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Right_Wall);
rcGenMaterials(1, &materialID[1]);
rcBindMaterial(materialID[1]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE,&Green.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Floor);
rcGenMaterials(1, &materialID[2]);
rcBindMaterial(materialID[2]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &White.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Ceiling);
rcGenMaterials(1, &materialID[3]);
rcBindMaterial(materialID[3]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &White.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Back_Wall);
rcGenMaterials(1, &materialID[4]);
rcBindMaterial(materialID[4]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &White.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcDisableClientState(RC_VERTEX_ARRAY);
rcStaticSceneEnd();
```

In the example, there are two boxes inside the background box. ***rcSceneAllInit()*** initializes static objects and sets basic initial parameters. And then, the background box is specified as a static object in between ***rcStaticSceneBegin()*** and ***rcStaticSceneEnd()***.

b. Scene Rendering

Scene rendering consists of two stages: defining lights and setting vertex data.

Four lights are on the ceiling. Each light can be switched on and off. When lights are not activated, a basic light is set automatically at the viewpoint.

```
rcMatrixMode(RC_MODELVIEW);
```

```
rcLoadIdentity();

rcRotatef(-g_fSpinY, 1.0f, 0.0f, 0.0f );
rcRotatef(-g_fSpinZ, 0.0f, 1.0f, 0.0f );

if (Light){
      rcEnable(RC_LIGHTING);
      rcLightfv(RC_LIGHT0, RC_AMBIENT, Light_Ambient);
      rcLightfv(RC_LIGHT0, RC_DIFFUSE, Light_Diffuse);
      rcLightfv(RC_LIGHT0, RC_POSITION, Light_Position1);
      rcEnable(RC_LIGHT0);


      rcLightfv(RC_LIGHT1, RC_AMBIENT, Light_Ambient);
      rcLightfv(RC_LIGHT1, RC_DIFFUSE, Light_Diffuse);
      rcLightfv(RC_LIGHT1, RC_POSITION, Light_Position2);
      rcEnable(RC_LIGHT1);

      rcLightfv(RC_LIGHT2, RC_AMBIENT, Light_Ambient);
      rcLightfv(RC_LIGHT2, RC_DIFFUSE, Light_Diffuse);
      rcLightfv(RC_LIGHT2, RC_POSITION, Light_Position3);
      rcEnable(RC_LIGHT2);

      rcLightfv(RC_LIGHT3, RC_AMBIENT, Light_Ambient);
      rcLightfv(RC_LIGHT3, RC_DIFFUSE, Light_Diffuse);
      rcLightfv(RC_LIGHT3, RC_POSITION, Light_Position4);
      rcEnable(RC_LIGHT3);
} else
      rcDisable(RC_LIGHTING);
```

c. Drawing Objects

There are two boxes. One is static, but the other is jumping. This type of dynamic object is processed in rendering loop, which is processed in an equivalently manner in OpenGL programming.

```
rcEnableClientState(RC_VERTEX_ARRAY);

rcPushMatrix();
{
      if (dir)
            aniy += 0.1;
      else
             aniy -= 0.1;

      if (aniy < 0) {
             aniy = 0;
             dir = true;
      } else if (aniy > 10) {
             aniy = 10;
             dir = false;
      }
      rcTranslatef(0, aniy, 0);
}

rcVertexPointer(3, RC_FLOAT, 0, Short_Block);
```

```
rcBindMaterial(MaterialID_ShortBlock);
rcColor4f(White1.r, White1.g, White1.b, 0);
rcDrawArrays(RC_TRIANGLES, 0, 30);
rcPopMatrix();

rcVertexPointer(3, RC_FLOAT, 0, Tall_Block);
rcBindMaterial(MaterialID_TallBlock);
rcColor4f(White2.r, White2.g, White2.b, 0);
rcDrawArrays(RC_TRIANGLES, 0, 30);

rcDisableClientState(RC_VERTEX_ARRAY);
```

d. Vertex Data

Vertex data is as follows:

```
Pos Left_Wall[] = {
      {55.28, 0.0, 0.0}, {54.96,    0.0, 55.92}, {55.60, 54.88, 55.92},
      {55.28, 0.0, 0.0}, {55.60, 54.88, 55.92}, {55.60, 54.88,     0.0}
};
Pos Right_Wall[] = {
      {0.0, 54.88, 0.0}, {0.0, 54.88, 55.92}, {0.0, 0.0, 55.92},
      {0.0, 54.88, 0.0}, {0.0,    0.0, 55.92}, {0.0, 0.0,     0.0}
};
Pos Floor[] = {
      {55.28, 0.0, 0.0}, {0.0, 0.0,    0.0}, {   0.0, 0.0, 55.92},
      {55.28, 0.0, 0.0}, {0.0, 0.0, 55.92}, {54.96, 0.0, 55.92}
};
Pos Ceiling[] = {
      {55.60, 54.88, 0.0}, {55.60, 54.88, 55.92}, {0.0, 54.88, 55.92},
      {55.60, 54.88, 0.0}, {   0.0, 54.88, 55.92}, {0.0, 54.88,     0.0}
};
Pos Back_Wall[] = {
      {0.0, 54.88, 55.92}, {55.60, 54.88, 55.92}, {54.96, 0.0, 55.92},
      {0.0, 54.88, 55.92}, {54.96,    0.0, 55.92}, {   0.0, 0.0, 55.92}
};
Pos Short_Block[] = {
      {13.0, 16.5,   6.5}, {  8.2, 16.50, 22.5}, {24.0, 16.5, 27.2},
      {13.0, 16.5,   6.5}, {24.0, 16.50, 27.2}, {29.0, 16.5, 11.4},
      {29.0,   0.0, 11.4}, {29.0, 16.50, 11.4}, {24.0, 16.5, 27.2},
      {29.0,   0.0, 11.4}, {24.0, 16.50, 27.2}, {24.0,   0.0, 27.2},
      {13.0,   0.0,   6.5}, {13.0, 16.50,   6.5}, {29.0, 16.5, 11.4},
      {13.0,   0.0,   6.5}, {29.0, 16.50, 11.4}, {29.0,   0.0, 11.4},
      { 8.2,   0.0, 22.5}, {  8.2, 16.50, 22.5}, {13.0, 16.0,   6.5},
      { 8.2,   0.0, 22.5}, {13.0, 16.50,   6.5}, {13.0,   0.0,   6.5},
      {24.0,   0.0, 27.2}, {24.0, 16.50, 27.2}, {8.20, 16.5, 22.5},
      {24.0,   0.0, 27.2}, {  8.2, 16.50, 22.5}, {8.20,   0.0, 22.5}
};
Pos Tall_Block[] = {
      {42.3, 33.0, 24.7}, {26.50, 33.0, 29.6}, {31.40, 33.0, 45.6},
      {42.3, 33.0, 24.7}, {31.40, 33.0, 45.6}, {47.20, 33.0, 40.6},
      {42.3,   0.0, 24.7}, {42.30, 33.0, 24.7}, {47.20, 33.0, 40.6},
      {42.3,   0.0, 24.7}, {47.20, 33.0, 40.6}, {47.20,   0.0, 40.6},
      {47.2,   0.0, 40.6}, {47.20, 33.0, 40.6}, {31.40, 33.0, 45.6},
      {47.2,   0.0, 40.6}, {31.40, 33.0, 45.6}, {31.40,   0.0, 45.6},
      {31.4,   0.0, 45.6}, {31.40, 33.0, 45.6}, {26.50, 33.0, 29.6},
      {31.4,   0.0, 45.6}, {26.50, 33.0, 29.6}, {26.50,   0.0, 29.6},
      {26.5,   0.0, 29.6}, {26.50, 33.0, 29.6}, {42.30, 33.0, 24.7},
      {26.5,   0.0, 29.6}, {42.30, 33.0, 24.7}, {42.30,   0.0, 24.7}
```

```
};
```

e. Rendering Result

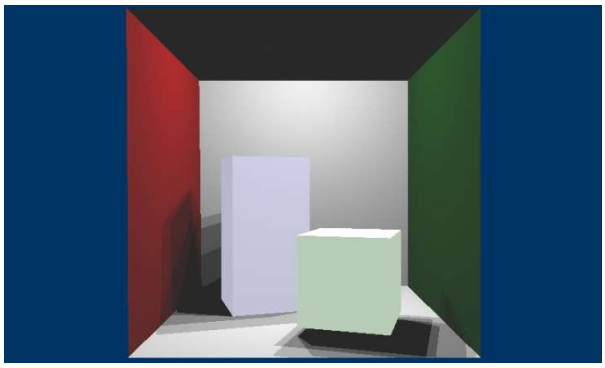

Figure 33. Rendering result

Appendix A.
Framework

```
#ifndef __EGL_FRAMEWORK_H__
#define __EGL_FRAMEWORK_H__
#ifdef WIN32
  #include "platform/WindowsPlatform.h"
  #ifndef RC_FRAMEWORK
        #pragma comment(lib, "RCFramework.lib")
  #endif
#else
  #include "platform/X11Platform.h"
#endif

class RCFramework :
  protected WindowPlatform
{
protected:
  virtual ~RCFramework(void);

public:
  RCFramework(const char* sTitleName, int width, int height);

  BOOL Initialize(void);
  void Release(void);
  BOOL DrawScene(void);

  inline EGLDisplay CurrentDisplay(void)  {return m_eglDisplay;}
  inline EGLContext CurrentContext(void)  {return m_eglContext;}
  inline EGLSurface CurrentSurface(void)  {return m_eglSurface;}
  inline int Width(void)          {return m_iWidth;}
  inline int Height(void)         {return m_iHeight;}
```

```
protected:
  virtual BOOL OnIntialize(void);
  virtual void OnRelease(void);
  virtual BOOL OnDraw(void);
  virtual BOOL OnPostDraw(void);

private:
  EGLDisplay        m_eglDisplay;
  EGLContext        m_eglContext;
  EGLSurface        m_eglSurface;
  int               m_iWidth, m_iHeight;
  const char*       m_sTitle;
};

#endif//__EGL_FRAMEWORK_H__
```

Appendix B.
Example Programs

RayCore® 1000 programming model consists of two stages – global initialization stage and data transfer stage. In global initialization stage, the global values such as frustum, view point and texture data which is used in rendering are specified. In data transfer stage, the data row that holds geometry transformation is transferred. Based on above gathered information, the final image is created using RayCore® 1000 hardware, a hardware which implements ray tracing algorithm. Data transfer stage is repeated in each scene, and through this process the control event is managed between the scenes.

1.1 Cube Object

This is a programming example of a cube with texture. As described in the first paragraph, programming is done in two stages.

a. Scene Initialization

In the initial setting stage, a couple of settings are made by RayCore® 1000 API.

The background color is set to black (0, 0, 0, 1). If the ray does not intersect with an object, the background color is stored in the frame buffer.

The depth test is not required in RayCore® 1000. Even if the depth test is set to activate, this will not affect the scene.

The projection matrix can be specified by frustum information. This step is same as that of OpenGL. Unlike in OpenGL where the view point is always set to origion, ray tracing can utilize any

coordinate. ***rcuLookAt*** function can be used to set the view point and this does not influence the projection matrix.

```
rcClearColor(0.0f, 0.0f, 0.0f, 1.0f);

rcMatrixMode(RC_PROJECTION);
rcLoadIdentity();
rcFrustumf(-0.9622504483f, 0.9622504483f, -0.577350269f, 0.577350269f, 1, 100);
rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);
```

b. Texture Load

Multiple textures can be used in the scenes. These textures are loaded prior to rendering. Initially, the texture objects are generated by using ***rcGenTextures***. Each texture ID is stored in array buffer. The current texture ID is specified by buffer ID before calling the texture loading function. Texture is loaded to nth texture object through current ID. All texture images are loaded by setting the current ID and by loading data. In this programming example, six textures are used in all sides of the cube.

Texture is used as mipmap by using ***rcTexImage*** function.

Texture is designed for one-time loading process because loading textures for every scene is a huge burden.

```
rcGenTextures(6, TextureArray);

rcBindTexture(RC_TEXTURE_2D, TextureArray[Blender]);
rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                    pBitmap[Blender]->sizeX, pBitmap[Blender]->sizeY,
                    0, RC_RGB, RC_UNSIGNED_BYTE,
                    pBitmap[Blender]->data);
```

c. Model

Model can be expressed in two ways. First is the index mode, and second is the sequential triangle mode. In the index mode, two arrays for indices and vertex coordinates are specified. The array of texture coordinates is added when using texture mapping. Also, the array of normal vectors can be used for vertex array.

Vertex position is stored in array. This is not used sequentially. The index number is allocated to this vertex.

```
Pos g_cubePos_indexed[] = {
      {-1.0f, -1.0f, 1.0f},      // 0
      {1.0f, -1.0f, 1.0f},       // 1
      {1.0f, 1.0f, 1.0f},        // 2
      {-1.0f, 1.0f, 1.0f},       // 3
      {-1.0f,-1.0f, -1.0f},      // 4
```

```
    {-1.0f, 1.0f, -1.0f},      // 5
    {1.0f, 1.0f, -1.0f},       // 6
    {1.0f,-1.0f, -1.0f},       // 7
};
```

Texture coordinates work in the same way as vertices do. However, because of its usage for 2D image, it is two dimensional. The first vertex uses the first texture coordinate.

```
TexC g_cubeTexcoord_indexed[] = {
    {0.0f, 0.0f},        // 0
    {1.0f, 0.0f},        // 1
    {1.0f, 1.0f},        // 2
    {0.0f, 1.0f},        // 3
    {0.0f, 0.0f},        // 4
    {0.0f, 1.0f},        // 5
    {1.0f, 1.0f},        // 6
    {1.0f, 0.0f},        // 7
};
```

The following shows the index array. The first four indices make a single rectangle and the next four indices make the next rectangle.

```
RCubyte g_cubeIndices[] =
{
    0, 1, 2, 3,     // Quad 0
    4, 5, 6, 7,     // Quad 1
    5, 3, 2, 6,     // Quad 2
    4, 7, 1, 0,     // Quad 3
    7, 6, 2, 1,     // Quad 4
    4, 0, 3, 5      // Quad 5
};
```

Sequential data mode is stored without indices. The same vertex is stored repeatedly; however, the rectangle is created sequentially as shown in the following code.

```
Pos g_cubePos[] =
{
    // Quad 0
    {-1.0f, -1.0f, 1.0f},      // 0 (unique)
    {1.0f, -1.0f, 1.0f},       // 1 (unique)
    {1.0f, 1.0f, 1.0f},        // 2 (unique)
    {-1.0f, 1.0f, 1.0f},       // 3 (unique)
    // Quad 1
    {-1.0f, -1.0f, -1.0f},     // 4 (unique)
    {-1.0f, 1.0f, -1.0f},      // 5 (unique)
    {1.0f, 1.0f, -1.0f},       // 6 (unique)
    {1.0f, -1.0f, -1.0f},      // 7 (unique)
    // Quad 2
    {-1.0f, 1.0f, -1.0f},      // 5 (start repeating here)
    {-1.0f, 1.0f, 1.0f},       // 3 (repeat of vertex 3)
```

```
        {1.0f, 1.0f, 1.0f},       // 2 (repeat of vertex 2... etc.)
        {1.0f, 1.0f, -1.0f},      // 6
        // Quad 3
        {-1.0f, -1.0f, -1.0f},    // 4
        {1.0f, -1.0f, -1.0f},     // 7
        {1.0f, -1.0f, 1.0f},      // 1
        {-1.0f, -1.0f, 1.0f},     // 0
        // Quad 4
        {1.0f, -1.0f, -1.0f},     // 7
        {1.0f, 1.0f, -1.0f},      // 6
        {1.0f, 1.0f, 1.0f},       // 2
        {1.0f, -1.0f, 1.0f},      // 1
        // Quad 5
        {-1.0f, -1.0f, -1.0f},    // 4
        {-1.0f, -1.0f, 1.0f},     // 0
        {-1.0f, 1.0f, 1.0f},      // 3
        {-1.0f, 1.0f,-1.0f}       // 5
};
```

d. Scene Rendering

The final image is generated by going through scene rendering process. This process is repeated until the program terminates. Three factors such as light setting, geometry transformation and dynamic object data needs to be specified.

Even if the light sources are not configured, RayCore® 1000 API sets one light source in view point. This is because the final color cannot be shown without light source. In this programming example, the light source is not set.

Geometry transformation is set just like the way it is done in OpenGL. Model view matrix is generated by using transformation functions such as ***rcTranslatef***, ***rcRotatef***, ***rcScalef***. This model view matrix is calculated based on the coordinates of each vertex. Vertex array and texture coordinates array is activated by using ***rcEnableClientState***.

```
rcMatrixMode( RC_MODELVIEW );
rcLoadIdentity();
rcTranslatef(0.0f, 0.0f, -6.0f);
rcRotatef(-g_fSpinY, 1.0f, 0.0f, 0.0f);
rcRotatef(-g_fSpinZ, 0.0f, 1.0f, 0.0f);
rcEnableClientState(RC_VERTEX_ARRAY);
rcEnableClientState(RC_TEXTURE_COORD_ARRAY);
```

rcVertexPointer connects the vertex array address. Texture mapping activated by using the parameter of **RC_TEXTURE_2D** is connected to material. Several parameters are transferred for their usage.

```
rcVertexPointer(3, RC_FLOAT, 0, g_cubePos_indexed);
rcTexCoordPointer(2, RC_FLOAT, 0, g_cubeTexcoord_indexed);

rcBindMaterial(MaterialIDs[Blender]);
```

```
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Blender]);
rcDrawElements(RC_TRIANGLE_FAN, 4, RC_UNSIGNED_BYTE, &g_cubeIndices[0] );

rcBindMaterial(MaterialIDs[Coffeematic]);
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Coffeematic]);
rcDrawElements(RC_TRIANGLE_FAN, 4, RC_UNSIGNED_BYTE, &g_cubeIndices[4] );
rcVertexPointer(3, RC_FLOAT, 0, g_cubePos);
rcTexCoordPointer(2, RC_FLOAT, 0, g_cubeTexs);

rcBindMaterial(MaterialIDs[Blender]);
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Blender]);
rcDrawArrays(RC_TRIANGLE_FAN, 0, 4);

rcBindMaterial(MaterialIDs[Coffeematic]);
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Coffeematic]);
rcDrawArrays(RC_TRIANGLE_FAN, 4, 4);
```

e. Rendering result

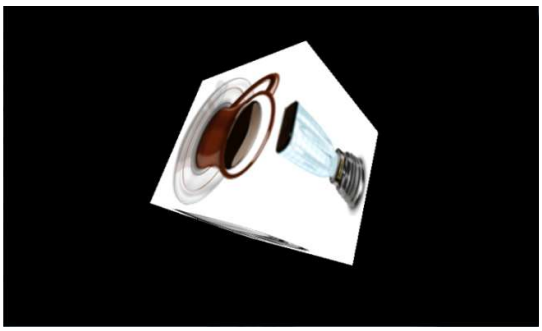

Figure 9. Rendering result

Appendix C.
Earth Example

Here is an example of light source movement.

```
#include "RCFramework.h"

// Programming Guide - Earth Example

#include "earth.h"

typedef struct RGBImageRec {
  int sizeX, sizeY;
  unsigned char *data;
} RGBImageRec;

typedef struct bmpBITMAPFILEHEADER{
  WORD    bfType;
  DWORD  bfSize;
  WORD    bfReserved1;
  WORD    bfReserved2;
  DWORD  bfOffBits;
} BITMAPFILEHEADER1;

typedef struct bmpBITMAPINFOHEADER{
  DWORD  biSize;
  DWORD  biWidth;
  DWORD  biHeight;
  WORD    biPlanes;
  WORD    biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  DWORD  biXPelsPerMeter;
  DWORD  biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPFILEHEADER2;

RGBImageRec *DIBImageLoad(char* path, int channel) {
  RGBImageRec* pImage=NULL;
  FILE *f=NULL;
  unsigned char *pBuf=NULL;
  int dataSize=0;
  int index=0;
  DWORD x=0;
  DWORD y=0;
  int bpp=0;
```

```
  if(channel !=3 && channel !=4) return pImage;

  f = fopen(path, "rb");
  if(f != NULL) {
        BITMAPFILEHEADER1 HD1;
        BITMAPFILEHEADER2 HD2;

        fseek(f, 0, SEEK_SET);
        fread(&HD1.bfType, sizeof(WORD), 1, f);
        fread(&HD1.bfSize, sizeof(WORD), 1, f);
        fseek(f, 10, SEEK_SET);
        fread(&HD1.bfOffBits, sizeof(int),1, f);

        fread(&HD2.biSize, sizeof(int), 1, f);
        fread(&HD2.biWidth, sizeof(int), 1, f);
        fread(&HD2.biHeight, sizeof(int), 1, f);
        fread(&HD2.biPlanes, sizeof(WORD), 1, f);
        fread(&HD2.biBitCount, sizeof(WORD), 1, f);
        fread(&HD2.biCompression, sizeof(int), 1, f);

        fseek(f,HD1.bfOffBits,SEEK_SET);

        bpp = HD2.biBitCount/8;
        if(bpp == 1 || bpp == channel)
        {
              pBuf = (unsigned char*) malloc(channel);

              pImage = (RGBImageRec*) malloc(sizeof(RGBImageRec));
              pImage->sizeX = HD2.biWidth;
              pImage->sizeY = HD2.biHeight;

              dataSize = HD2.biWidth*HD2.biHeight*channel;
              pImage->data = (unsigned char*) malloc(dataSize);

              for(y=0; y<HD2.biHeight; y++) {
                    for(x=0; x<HD2.biWidth; x++) {
                          fread(pBuf, bpp, 1, f);
                          if(bpp == 1) {
                                pBuf[1] = pBuf[2] = pBuf[0];
                                if(channel == 4)       pBuf[3] = 0;
                          }

                          index = (y*HD2.biWidth + x)*channel;
                          pImage->data[index]       = pBuf[2];
                          pImage->data[index + 1]  = pBuf[1];
                          pImage->data[index + 2]  = pBuf[0];
                          if(channel == 4)
                                pImage->data[index + 3] = pBuf[3];
                    }
              }

              if(pBuf) free(pBuf);
              pBuf = NULL;
        }

        fclose(f);
  }

  return pImage;
}

#define Galaxy            0
#define Earth             1

unsigned int TextureArray[2];
unsigned int MaterialIDs[2];

float Light_Position[] = {-17.802584, 0.353599, 8.770458, 1};
float Light_Ambient[] = {0, 0, 0};
float Light_Diffuse[] = {1, 1, 1};
float Light_Specular[] = {1, 1, 1};

float g_turn = 35;

class Tutorial : public RCFramework
```

```
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Earth Example", 800, 480){}
  virtual ~Tutorial(void){}

  void Load(RGBImageRec **pBitmap){

        pBitmap[0] = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Earth/Galaxy.bmp", 3);
        pBitmap[1] = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Earth/Earth.bmp", 3);
  }

  void CreateTexture(void){
        RGBImageRec *pBitmap[2];
        int i=0;

        for(i=0; i<2; i++) {
              pBitmap[i] = NULL;
        }
        Load(pBitmap);

        rcGenTextures(2, TextureArray);

        rcBindTexture(RC_TEXTURE_2D, TextureArray[Galaxy]);
        rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                           pBitmap[Galaxy]->sizeX, pBitmap[Galaxy]->sizeY,
                           0, RC_RGB, RC_UNSIGNED_BYTE,
                           pBitmap[Galaxy]->data);

        rcBindTexture(RC_TEXTURE_2D, TextureArray[Earth]);
        rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                           pBitmap[Earth]->sizeX, pBitmap[Earth]->sizeY,
                           0, RC_RGB, RC_UNSIGNED_BYTE,
                           pBitmap[Earth]->data);

        for(i=0; i<2; i++) {
              if(pBitmap[i]) {
                    if(pBitmap[i]->data) free(pBitmap[i]->data);
                    free(pBitmap[i]);
              }
        }
  }

  void SetMaterial(void){
        float AmbientDiffuse[2][3] = {
              {0.588235, 0.588235, 0.588235},
              {0.529412, 0.529412, 0.529412},
        };

        float Specular[2][3] = {
              {0, 0, 0},
              {1, 0, 0},
        };

        float Shininess[2] = {
              2.000000,
              3.031433,
        };

        rcEnable(RC_TEXTURE_2D);

        rcGenMaterials(1, &MaterialIDs[Galaxy]);
        rcBindMaterial(MaterialIDs[Galaxy]);
        rcBindTexture(RC_TEXTURE_2D, TextureArray[Galaxy]);
        rcMaterialfv(RC_FRONT_AND_BACK,
                           RC_AMBIENT_AND_DIFFUSE,
                           AmbientDiffuse[Galaxy]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, Specular[Galaxy]);
        rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, Shininess[Galaxy]);

        rcGenMaterials(1, &MaterialIDs[Earth]);
        rcBindMaterial(MaterialIDs[Earth]);
        rcBindTexture(RC_TEXTURE_2D, TextureArray[Earth]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, AmbientDiffuse[Earth]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, Specular[Earth]);
        rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, Shininess[Earth]);
  }
```

```
  void StaticSceneDraw(void){
        RCuint materialID;

            float galaxyV[12] = {
                  -1, -1,   0,
                   1, -1,   0,
                   1,  1,   0,
                  -1,  1,   0
            };

            float galaxyT[8] = {
                  0, 0,
                  1, 0,
                  1, 1,
                  0, 1
            };

  rcBindMaterial(MaterialIDs[Galaxy]);
  rcEnable(RC_TEXTURE_2D);
  rcBindTexture(RC_TEXTURE_2D, TextureArray[Galaxy]);

  rcStaticSceneBegin();

  rcEnableClientState(RC_VERTEX_ARRAY);
  rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

  rcVertexPointer(3, RC_FLOAT, 0, galaxyV);
  rcTexCoordPointer(2, RC_FLOAT, 0, galaxyT);

  rcMatrixMode(RC_MODELVIEW);
  rcLoadIdentity();
  rcTranslatef(0, 0, -18.0);
  rcScalef(8, 4.8, 1);

  rcEnable(RC_USE_TEXTURE_ONLY);
  rcDrawArrays(RC_QUADS, 0, 4);
  rcDisable(RC_USE_TEXTURE_ONLY);

  rcDisableClientState(RC_VERTEX_ARRAY);
  rcDisableClientState(RC_TEXTURE_COORD_ARRAY);

  rcStaticSceneEnd();
 }

protected:
  virtual BOOL OnIntialize(void){

rcDepthBounce(14);

rcSceneAllInit();

        {
            rcClearColor(0.1f, 0.1f, 0.1f, 1.0f);
            rcViewport(0, 0, Width(), Height());

            rcMatrixMode(RC_PROJECTION);
            rcLoadIdentity();

            rcuPerspective(30, (float)Width() /(float)Height(), 10, 10000 );

            rcuLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);

            CreateTexture();
            SetMaterial();

            {
                  rcEnable(RC_LIGHTING);
                  rcEnable(RC_LIGHT0);
                  rcLightfv(RC_LIGHT0, RC_POSITION, Light_Position);
                  rcLightfv(RC_LIGHT0, RC_AMBIENT, Light_Ambient);
                  rcLightfv(RC_LIGHT0, RC_DIFFUSE, Light_Diffuse);
                  rcLightfv(RC_LIGHT0, RC_SPECULAR, Light_Specular);
            }

            StaticSceneDraw();
```

```
        }

        return TRUE;
    }

    virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        {
            rcClear(RC_COLOR_BUFFER_BIT | RC_DEPTH_BUFFER_BIT);

            rcMatrixMode(RC_MODELVIEW);
            rcLoadIdentity();

            rcPushMatrix();
            rcRotatef(g_turn, 0, 1, 0);
            rcLightfv(RC_LIGHT0, RC_POSITION, Light_Position);
            rcPopMatrix();

            rcTranslatef(0, 0, -7.0);
            rcRotatef(35, 0, 1, 0);
            rcRotatef(15, 1, 0, 0);
            rcRotatef(-90, 0, 0, 1);

            rcEnableClientState(RC_VERTEX_ARRAY);
            rcEnableClientState(RC_NORMAL_ARRAY);
            rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

            rcVertexPointer(3, RC_FLOAT, sizeof(struct Pos), g_EarthVertices);
            rcNormalPointer(RC_FLOAT, 0, g_EarthNormals);
            rcTexCoordPointer(2, RC_FLOAT, 0, g_EarthTexCoords);

            rcBindMaterial(MaterialIDs[Earth]);
            rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_EarthVertices)/sizeof(Pos));

            rcDisableClientState(RC_VERTEX_ARRAY);
            rcDisableClientState(RC_NORMAL_ARRAY);
             rcDisableClientState(RC_TEXTURE_COORD_ARRAY);

            g_turn += 5;

            if (g_turn == 360)
                g_turn = 0;
        }

        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

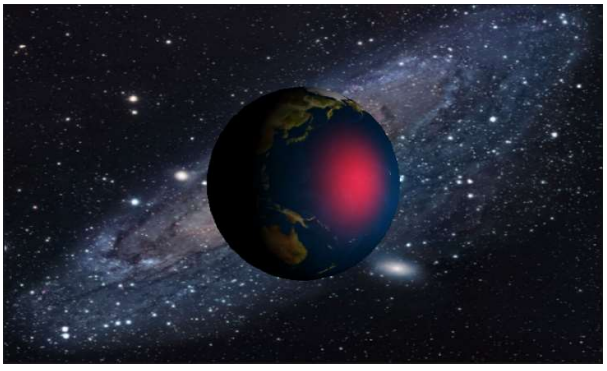

Figure 10. Rendering result

Appendix D.
Program Development Environment Configuration

1.1 Development Environment Configuration for Linux

a. Path Configuration for Library

The libaries which are required for RayCore® 1000 rendering are located in the following directory:
‘~/Demos/app/lib’s

These libraries should be compiled after configurating its path in symbolic link or ‘/etc’ folder.

- Set the folder of libraries that contains provided SDK while writing Makefile.
  (e.g.)
  32bit linux: ‘~/Demos/app/lib/linux_x86’,
  64bit linux: ‘~/Demos /app/lib/linux_x68’
  s
  (e.g.) LIBDIR := $(LIBDIR) -lstdc++ -lpthread -lm -lX11 lm -lX11 –lRayCoreAPI -lRCDDK –lRCHAL –lRCDDIArriaV

- A static libary of ‘RCFramework.a’ is generated in ‘~/RayCoreSDK_Altera_ArriaV/app/lib’ directory when makefile is compiled
- Following are the necessary libraries for application compiling.

[Libraries included in Linux]

stdc++, pthread , m, X11 (It is recommended that libgtk2.0-dev is installed in Xlibrary.)

[Dynamic and static libraries of RayCore® 1000]

libRayCoreAPI.so, libRCDDK.so, libRCHAL.so, libRCDDIArriaV.so, RCFramework.a

b. Header File Path Configuration

In order to contain 'include path' when writing Makefile, the configuration should be done as follows.

```
INC    := \
          -I$(path that contains example file source code) \
          -I../RCFramework/src       \
          -I../../../include \
          -I../../../include/khronos    \
          -I../../../include/siliconarts
```

c. Example of Makefile

```
## Copyright (c) | 2010 ~ 2013 Siliconarts, Inc. All Rights Reserved
## tutorials
## Date :

.SUFFIXES : .cc .o

CXX                    := $(CROSS)g++

INC       := \
          -I../RCFramework/src       \
          -I../../../include \
          -I../../../include/khronos    \
          -I../../../include/siliconarts

ifeq ($(LBITS),64)
  OS_ARCH      := linux_x64
else
  OS_ARCH      := linux_x86
endif

LIBS   := -L../../../lib/$(OS_ARCH)
LIBS       := $(LIBS) -lstdc++ -lpthread -lm -lX11 -lRayCoreAPI -lRCDDK -lRCHAL -lRCDDIArriaV
CXXFLAGS       = $(INC)
SRC_PATH               := $(PWD)

OBJS       := \
  $(SRC_PATH)/main.o

SRCS                   := \
  $(SRC_PATH)/main.cpp

TARGET       = rc_simpletriangle

all : $(TARGET)

$(TARGET) : $(OBJS)
  $(CXX) -o $@ ../../../lib/RCFramework.a $(OBJS) $(LIBS)

dep :
  gccmakedep $(INC) $(SRCS)

clean :
  rm -rf $(OBJS) $(TARGET) core

new :
  $(MAKE) clean
  $(MAKE)
```

1.2 EGL Native Windows Configuration

Because RayCore® 1000 Development Kit support Linux, EGL is platform-dependent based on each characteristic.

a. Linux

EGL uses header files such as X11/Xlib.h, X11/Xutil.h, X11/Xatom.h in Linux. Window handle in Linux's X library is called to generate a window.

Figure 11. Application window generation in Linux

1.3 EGL Configuration

EGL supports interface functions between operating systems such as Linux and RayCore® 1000 API. EGL plays a role of generating the Surface after receiving the hardware display information and of printing out the rendered scenes on the Context. In case of utilizing RayCore® 1000 Development Kit to develop applications, EGL setting is required at 'RCFramework.a' which is a static library. Also, ***rcSceneAllInit*** function should be called to generate and initialize EGL in the initial program stage.

a. EGL Initiazliation

EGL initialization should be done as follows: the attributes are set to 8, 8, 8 which is the basic RGB. DEPTH is set to 16. EGL_RENDERABLE_TYPE is set to EGL_OPEN_EG_BIT. EGL EGL_CONTEXT_CLIENT_VERSION should be set to 1 for attribute context.

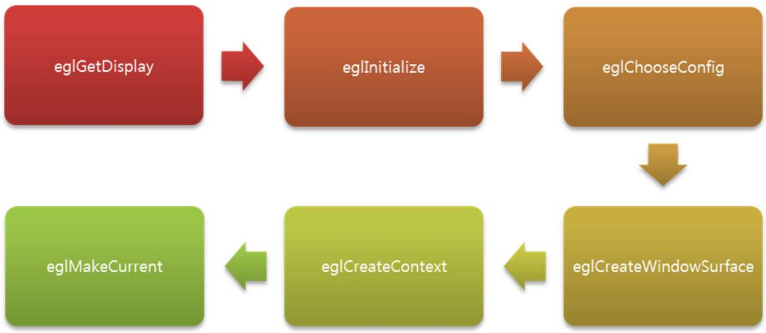

Figure 12. EGL initialization flow chart

```
EGLint      attribConfig[] = {
        EGL_RED_SIZE,          8,
        EGL_GREEN_SIZE,        8,
        EGL_BLUE_SIZE,         8,
        EGL_DEPTH_SIZE,        16,
        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
        EGL_NONE
};

EGLint      attribContext[] = {
        EGL_CONTEXT_CLIENT_VERSION, 1,
        EGL_NONE
};
```

b. Drawing EGL

Generally, ***eglSwapBufferis*** is called after calling ***rcFinish***. In case ***rcFinish*** has not been called, ***eglSwapBuffer*** calls this function internally. Therefore, only ***eglSwapBuffer*** may be called.

```
rcFinish();
eglSwapBuffers(eglDisplay, eglSurface);
```

c. Terminating EGL

EGL is terminated by calling eglMakeCurrent(eglDisplay,EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT.

1.4 Programming using RCFramework Class

a. General Setting

'RCFramework.h' header needs to be included. Since RCFrameWork static library has EGL declaration to be suitable for RayCore® 1000 API, this library must be utilized in the configuration as well as generation and implementation of display, surface, and context value. RayCore® 1000 API is a status machine with which a user is required to set the status, using the following API, to print out the rendering scenes.

b. Initialization

OnInitialize(void) is a virtual function that checks the status of initialization by returning *true* or *false* signal. The initial setting of RayCore® 1000 API is done using this function.

```
rcSceneAllInit(); //Initialize both static and dynamic scene data

//Set the RayCore    API state variables
rcClearColor(0.0f, 0.0f, 0.0f, 1.0f);      //Set the clear color
rcViewport(0, 0, Width(), Height());    //Set the viewport by window size
rcMatrixMode(RC_PROJECTION);    //Set the projection mode
rcLoadIdentity();
rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);

StaticSceneDraw();                          //Draw the static scene
```

The binding parts should be released if the material and texture are generated and used.

```
rcBindMaterial(0);
rcBindTexture(RC_TEXTURE_2D, 0);
rcDisable(RC_TEXTURE_2D);
```

c. Draw Scene Data

Before and after drawing static scenes, ***rcStaticSceneBegin()*** and ***rcStaticSceneEnd()*** functions should be configured. After static scenes is set to ***OnInitialize(void)*** in the beginning, the static scenes are always depicted on the screen even if the screen refreshes.

All other data that are not set as static scenes are drawn as dynamic scenes. New setting is always necessary using ***OnDraw(void)*** before the screen refresh in order to depict the scenes.

The following is the sequence for simple scene rendering.

- Matrix mode setting

```
rcMatrixMode(RC_MODELVIEW);
rcLoadIdentity();
```

- Material binding and setting

```
rcBindMaterial(1);
rcMaterialfv(RC_FRONT_AND_BACK, C_AMBIENT_AND_DIFFUSE, color);
```

- Vertex drawing

```
rcEnableClientState(RC_VERTEX_ARRAY);
rcVertexPointer(3, RC_FLOAT, 0, vertices);
rcDrawArrays(RC_TRIANGLES, 0, 3);
rcDisableClientState(RC_VERTEX_ARRAY);
```

1.5 General Programming Source Code

```
//*****************************//
//Example : Simple Reflection          //
//*****************************//
EGLDisplay m_eglDisplay;
EGLContext m_eglContext;
EGLSurface m_eglSurface;
int             m_iWidth = 800;
int             m_iHeight = 480;

#define MATERIAL_FRAME          0
#define MATERIAL_MIRROR         1
```

```
#define MATERIAL_TRIANGLE     2
unsigned int g_MaterialArray[3];

RCfloat       m_Angle=0.0f;

int main(int argc, char ** argv) {

      //****************************//
      //Initialize EGL
      //****************************//
      {
            EGLint       attribConfig[] = {
                  EGL_RED_SIZE,        8,
                  EGL_GREEN_SIZE,   8,
                  EGL_BLUE_SIZE,      8,
                  EGL_DEPTH_SIZE,    16,
                  EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
                  EGL_NONE
            };
            EGLint       attribContext[] = {
                  EGL_CONTEXT_CLIENT_VERSION, 1,
                  EGL_NONE
            };
            EGLConfig eglConfig;
            EGLint nConfigs = 0;

            m_eglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf("There is an error while the function, eglGetDisplay.\n");
                  return 0;
            }
            eglInitialize(m_eglDisplay, NULL, NULL);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf("There is an error while the function, eglInitialize.\n");
                  return 0;
            }
            eglChooseConfig(m_eglDisplay, attribConfig, &eglConfig, 1, &nConfigs);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf("There is an error while the function, eglChooseConfig.\n");
                  return 0;
            }
            if(nConfigs == 0)
            {
                  printf("There is no config selected.\n");
                  return 0;
            }

            //HWND HWnd;
            //m_eglSurface = eglCreateWindowSurface(m_eglDisplay, eglConfig,
(EGLNativeWindowType)HWnd, 0);
            m_eglSurface = eglCreateWindowSurface(m_eglDisplay, eglConfig, NULL, 0);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf(   "There is an error while the function, eglCreateWindowSurface.\n");
                  return 0;
            }
            m_eglContext = eglCreateContext(m_eglDisplay, eglConfig, EGL_NO_CONTEXT, attribContext);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf("There is an error while the function, eglCreateContext.\n");
                  return 0;
            }
            if(eglMakeCurrent(m_eglDisplay, m_eglSurface, m_eglSurface, m_eglContext) != EGL_TRUE)
            {
                  printf("There is an error while the function, eglMakeCurrent.\n");
                  return 0;
            }
      }
      //****************************//


      //****************************//
      //Set the camera
```

```
//*****************************//
{
  //Set the clear color
    rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);

    //Set the viewport by window size
    rcViewport(0, 0, m_iWidth, m_iHeight);

    rcMatrixMode(RC_PROJECTION);
    rcLoadIdentity();

    rcuPerspective(60.0f, (RCfloat)m_iWidth / (RCfloat)m_iHeight, 1.0f, 500.0f);
    rcuLookAt(-3, 0, 0, 0, 0, -8, 0, 1, 0);
}
//*****************************//


//*****************************//
//Set the light
//*****************************//
{
    RCfloat ambient[] = {0.2f, 0.2f, 0.2f, 1.0f};
    RCfloat diffuse[] = {0.7f, 0.7f, 0.7f, 1.0f};
    RCfloat specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    RCfloat pos[] = {0.0f, -0.5f, 1.0f, 1.0f};

    rcLightfv(RC_LIGHT0, RC_POSITION, pos);
    rcLightfv(RC_LIGHT0, RC_AMBIENT, ambient);
    rcLightfv(RC_LIGHT0, RC_DIFFUSE, diffuse);
    rcLightfv(RC_LIGHT0, RC_SPECULAR, specular);
}
//*****************************//


//*****************************//
//Set the material
//*****************************//
{
    RCfloat Reflectance[] = {1.0, 1.0, 1.0};
    RCfloat NotReflectance[] = {0, 0, 0};
    RCfloat color[3][4]={
        {0.2f, 0.4f, 0.8f, 0.0f},
        {0.0f, 0.8f, 0.4f, 0.0f},
        {0.0f, 1.0f, 0.0f, 0.0f},
    };

    rcGenMaterials(3, g_MaterialArray);

    rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);
    rcDisable(RC_TEXTURE_2D);
    rcMaterialfv(RC_FRONT_AND_BACK,
                 RC_AMBIENT_AND_DIFFUSE,
                 &color[MATERIAL_FRAME][0]);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);

    rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);
    rcDisable(RC_TEXTURE_2D);
    rcMaterialfv(RC_FRONT_AND_BACK,
                 RC_AMBIENT_AND_DIFFUSE,
                 &color[MATERIAL_MIRROR][0]);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, Reflectance);

    rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
    rcDisable(RC_TEXTURE_2D);
    rcMaterialfv(RC_FRONT_AND_BACK,
                 RC_AMBIENT_AND_DIFFUSE,
                 &color[MATERIAL_TRIANGLE][0]);

    RCfloat ambient[] = {0.8,0.8,0.8,0};
    RCfloat diffuse[] = {0.2,0.2,0,0};
    rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
    rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);

    rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);
}
//*****************************//
```

```
//*****************************//
//Initialize all scene
//*****************************//
//Initialize both static and dynamic scene data
rcSceneAllInit();
//*****************************//


//*****************************//
//Draw the static scene
//*****************************//
rcStaticSceneBegin();
{
    RCfloat QuadVertices[4][3];

    //Rectangle
    QuadVertices[0][0] = 2.0f;   QuadVertices[0][1] = -2.0f;    QuadVertices[0][2] = 0.0f;
    QuadVertices[1][0] = 2.0f;   QuadVertices[1][1] = 2.0f;     QuadVertices[1][2] = 0.0f;
    QuadVertices[2][0] = -2.0f;  QuadVertices[2][1] = 2.0f;     QuadVertices[2][2] = 0.0f;
    QuadVertices[3][0] = -2.0f;  QuadVertices[3][1] = -2.0f;    QuadVertices[3][2] = 0.0f;

    rcDisable(RC_TEXTURE_2D);

    rcEnableClientState(RC_VERTEX_ARRAY);

    //Set the modelview matrix mode
    rcMatrixMode(RC_MODELVIEW);
    rcLoadIdentity();

    //Set the vertex data
    rcVertexPointer(3, RC_FLOAT, 0, QuadVertices);


    //Bind the material of a mirror's frame
    rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);

    //Rendering the mirror's frame
    rcPushMatrix();
    rcTranslatef(0, 0, -10);
    rcScalef(2, 2, 2);
    rcDrawArrays(RC_QUADS,0,4);
    rcPopMatrix();


    //Bind the material of a mirror
    rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);

    //Rendering the mirror
    rcPushMatrix();
    rcTranslatef(0, 0, -9.999f);
    rcScalef(1.7f, 1.7f, 1.7f);
    rcDrawArrays(RC_QUADS, 0, 4);
    rcPopMatrix();

    rcDisableClientState(RC_VERTEX_ARRAY);
}
rcStaticSceneEnd();
//*****************************//


//*****************************//
//Draw the dynamic scene
//*****************************//
{
    RCfloat TriangleVertices[3][3];

    //Triangle
    TriangleVertices[0][0] = 0.0f;     TriangleVertices[0][1] = 0.707f;     TriangleVertices[0][2] =
0.0f;
    TriangleVertices[1][0] = -1.0f;    TriangleVertices[1][1] = -0.707f;    TriangleVertices[1][2] =
0.0f;
    TriangleVertices[2][0] = 1.0f;     TriangleVertices[2][1] = -0.707f;    TriangleVertices[2][2] =
0.0f;
```

```
        while(1)
        {
            rcEnableClientState(RC_VERTEX_ARRAY);

            //Set the modelview matrix mode
            rcMatrixMode(RC_MODELVIEW);
            rcLoadIdentity();

            //Set the vertex data
            rcVertexPointer(3, RC_FLOAT, 0, TriangleVertices);

            //Bind the material of a triangle
            rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);

            //Rendering the triangle
            rcPushMatrix();
            rcTranslatef(0, 0, -5);
            rcRotatef(m_Angle, 0, 1, 0);
            rcDrawArrays(RC_TRIANGLES, 0, 3);
            rcPopMatrix();

            rcDisableClientState(RC_VERTEX_ARRAY);

            //Display all scene on the screen
            if(rcGetError() == RC_NO_ERROR)
            {
                rcFinish();
                eglSwapBuffers(m_eglDisplay, m_eglSurface);
            }else{
                printf("There was an error while drawing scene.\n");
                break;
            }

            //Change the rotation angle of a triangle
            m_Angle += 10.f;
        }
    }
    //*****************************//


    //*****************************//
    //Release EGL
    //*****************************//
    if(m_eglDisplay) eglMakeCurrent(m_eglDisplay,
                                    EGL_NO_SURFACE,
                                    EGL_NO_SURFACE,
                                    EGL_NO_CONTEXT);
    if(m_eglSurface != EGL_NO_SURFACE)  eglDestroySurface(m_eglDisplay, m_eglSurface);
    if(m_eglContext != EGL_NO_CONTEXT)  eglDestroyContext(m_eglDisplay, m_eglContext);
    if(m_eglDisplay != EGL_NO_DISPLAY)  eglTerminate(m_eglDisplay);
    m_eglDisplay    = EGL_NO_DISPLAY;
    m_eglContext    = EGL_NO_CONTEXT;
    m_eglSurface    = EGL_NO_SURFACE;
    //*****************************//

}
```