# Computer Graphics and Programming

## Lecture 10
## 3D Graphics Class for OpenGL

Jeong-Yean Yang

2020/12/8



# 1 Integration of OpenGL Class

# Step 1. Multiple Object Management

- Open has Three types of objects
- 1. Object
  - Box, cylinder, sphere, plane, and so on
  - Vertex buffer: Position, Normal, UV vectors
  - Element buffer: face(or polygon)
- 2. Shader
  - Phong shading, solid shading, and so on
  - GLSL based vertex (.vsh) and fragment (.fsh) script
- 3. Texture
  - Image (png or jpg) is mapped onto vertex by UV value

- We need to handle multiple and dynamic objects



# Create Object and Close Object

- Principles of Programming for memory allocation
  - Create(init or new or alloc )  Vs Destroy(close or delete or free)
  - create(memory allocation)
  - close(delete memory block)
  - If you miss “close”, memory leakage occurs → **critical problems**

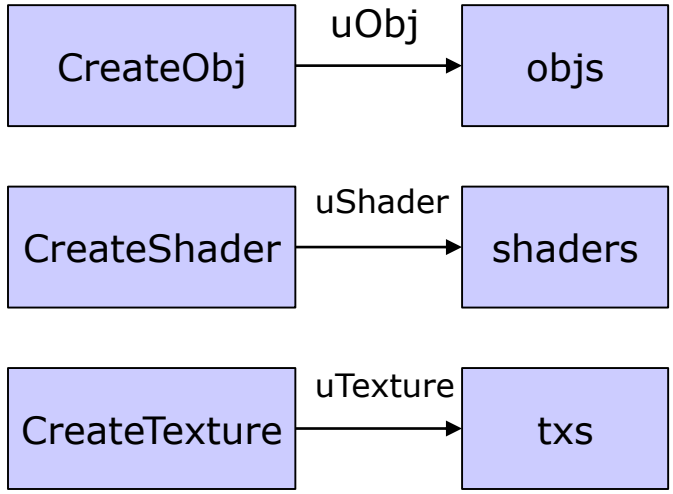
- uObj::Alloc(create) : create vertex, normal, UV, and VBO
- uObj::Close: delete vertex, normal, UV, and VBO
- uShader::Load(create) : create GLSL program
- uShader:Close: delete GLSL program
- uTexture::Load(create): create texture buffer
- uTexture::Close: delete texture buffer



# Dynamic Array for Object Management

## Ex) uGL-26-Complete-GL objects

```
class uWnd : public uGL
{
public:
        uWnd();
public:
        // Basic GL functions
        virtual void Draw();
        virtual void Loading();
        virtual void Run();
        virtual void Close();

        // GL object
        uObj*       CreateObj();
        uShader*    CreateShader(char *vsh,char *fsh);
        uTexture*   CreateTexture(char *imgname);

public:
        vArray<uObj*,uObj*>           objs;
        vArray<uShader*,uShader*>     shaders;
        vArray<uTexture*,uTexture*>   txs;
```




- Each object is stored in Dynamic array, vArray.

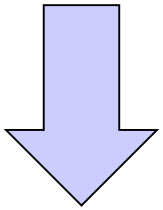


# New OpenGL Object Management "Create()"

```
// object creation
uObj *p = CreateObj();
p->MakeBox(10,10,0.1);
...
p->Update();
```

```
uObj* uWnd::CreateObj()
{
    uObj* pNew = new uObj();
    objs.Add(pNew);
    return pNew;
}
```

```
public:
        vArray<uObj*,uObj*>          objs;
```



```
// object creation
uObj *p = CreateObj();
p->MakeBox(10,10,0.1);
...
p->Update();
```

```
void uObj::MakeBox(float a,float b,float c)
{
    Alloc(8,12);

    pVer[0].v = uVector(0,0,0);
    pVer[1].v = uVector(a,0,0);
```

```
void uObj::Alloc(int nv,int np)
{
    Close();

    nVer    = nv;
    nPoly   = np;

    pVer    = new uVertex[nv];
```



# New OpenGL Object Management "Close()"

```
void uWnd::Close()
{
    int i;

    // remove all objects
    for (i=0;i<objs.GetSize();i++)
    {
        uObj *p = objs[i];
        p->Close();
        delete p;
    }
    objs.RemoveAll();

    // remove all shaders
    for (i=0;i<shaders.GetSize();i++)
    {
        uShader *p = shaders[i];
        p->Close();
        delete p;
    }
    shaders.RemoveAll();

    // remove all textures
    for (i=0;i<txs.GetSize();i++)
    {
        uTexture *p = txs[i];
        p->Close();
        delete p;
    }
    txs.RemoveAll();
}
```

```
public:
        vArray<uObj*,uObj*>          objs;
```

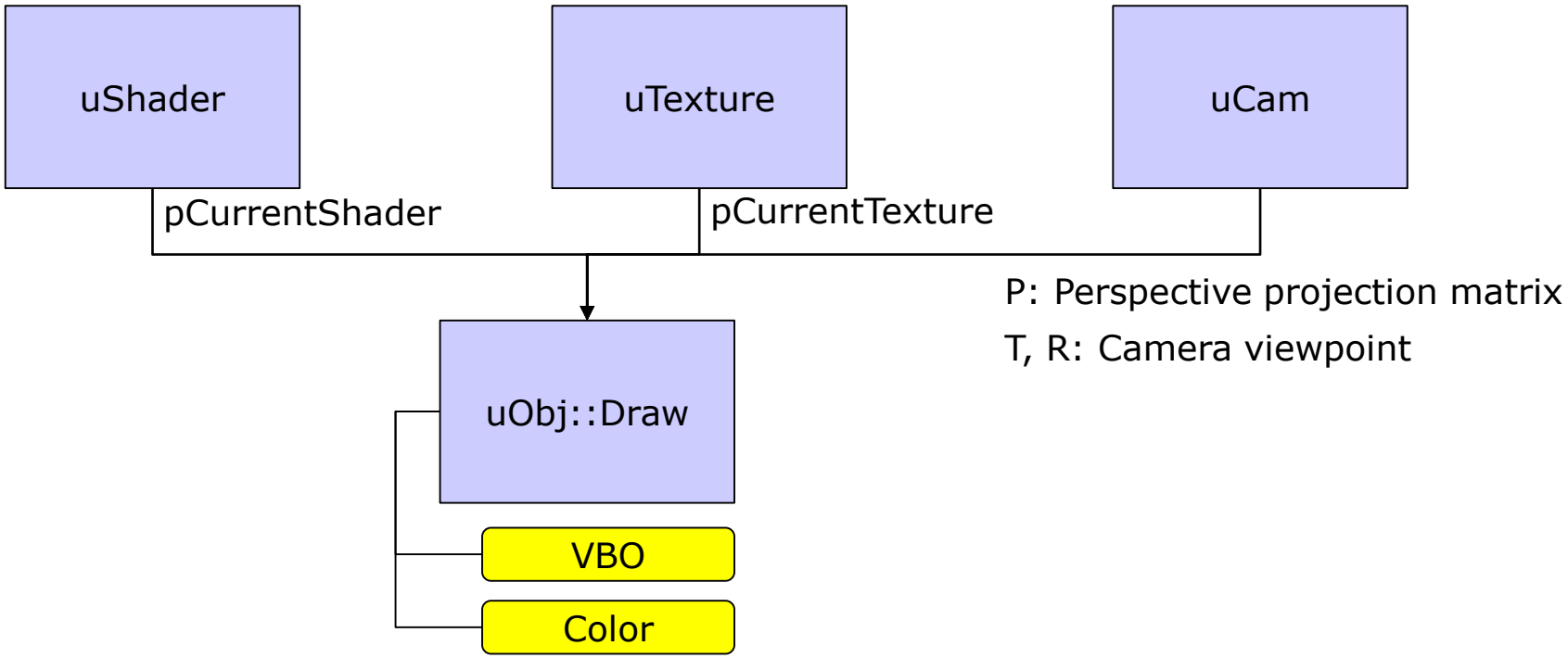
```
void uObj::Close()
{
    if (pVer)    delete pVer;
    if (pPoly)   delete pPoly;
    if (pTemp)   delete pTemp;

    pVer    = NULL;
    pPoly   = NULL;
    pTemp   = NULL;

    if (vs) glDeleteBuffers(1, &vs);     vs   = 0;
    if (fs) glDeleteBuffers(1, &fs);     fs   = 0;
}
```

Delete vertex memory in CPU

Delete VBO in GPU



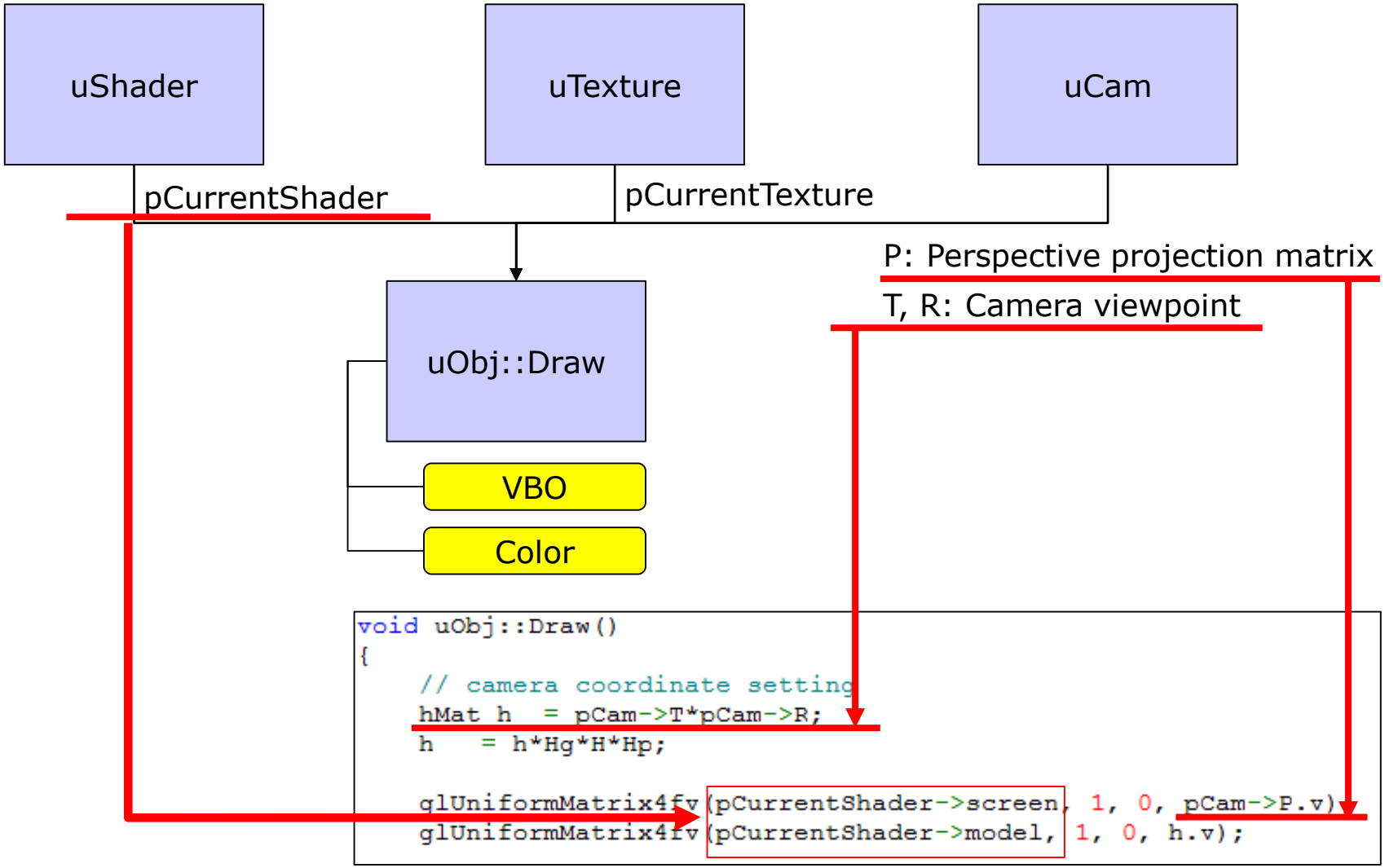
# Step 2: Draw() in a New Structure



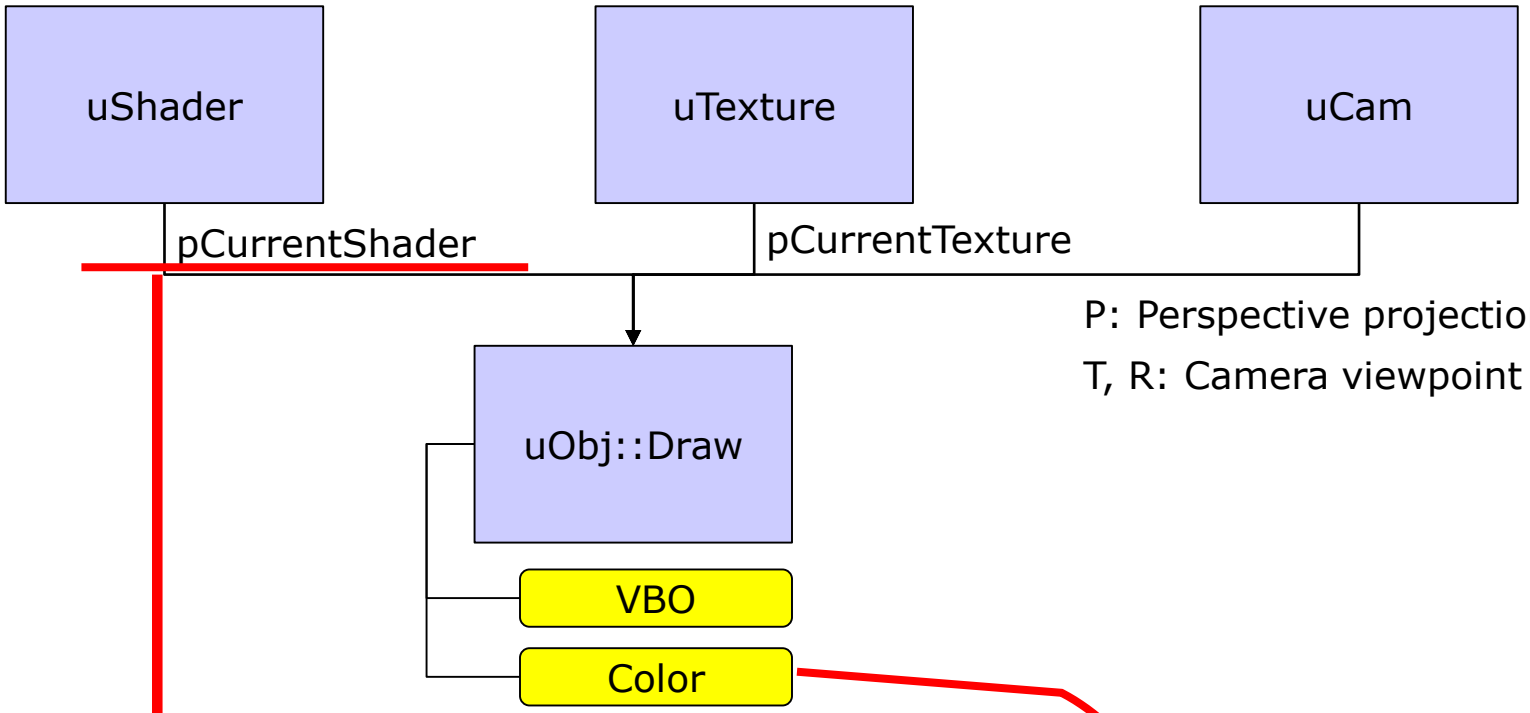

void uObj::Draw()



# Step 2: Draw() in a New Structure
# Step 2-1: Camera



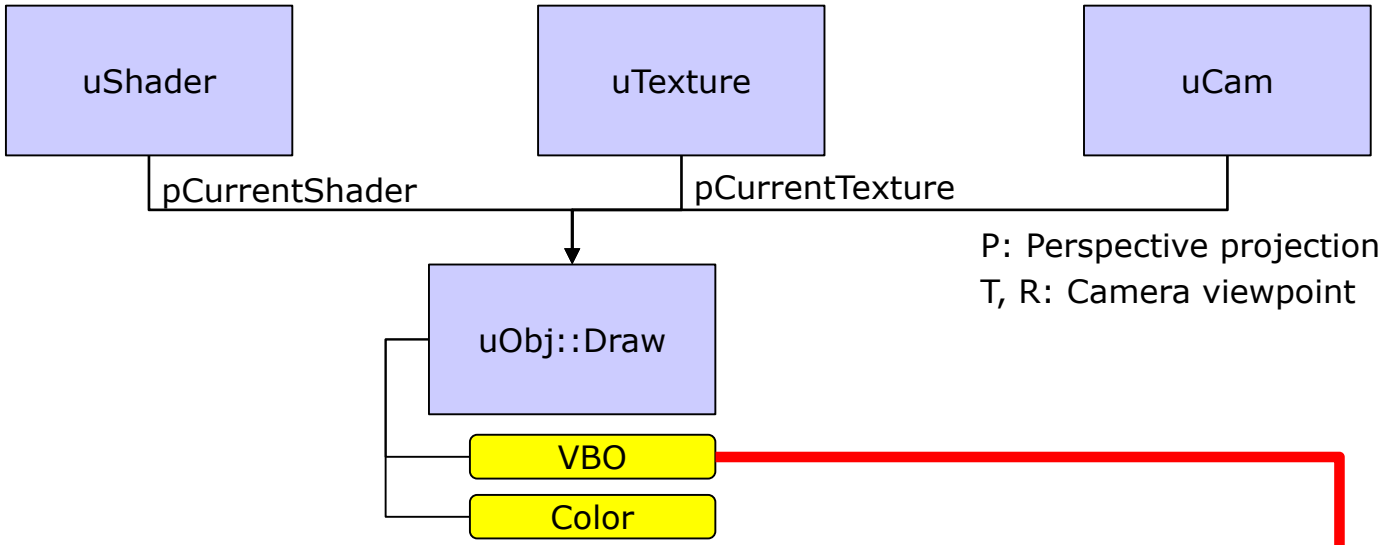
# Step 2: Draw() in a New Structure
# Step 2-2: Colors




```
// set colors
glUniform4f(pCurrentShader->diffuse, diffuse.r,diffuse.g,diffuse.b,diffuse.a);
glUniform4f(pCurrentShader->ambient, ambient.r,ambient.g,ambient.b,ambient.a);
glUniform4f(pCurrentShader->specular, specular.r,specular.g,specular.b,specular.a);
```



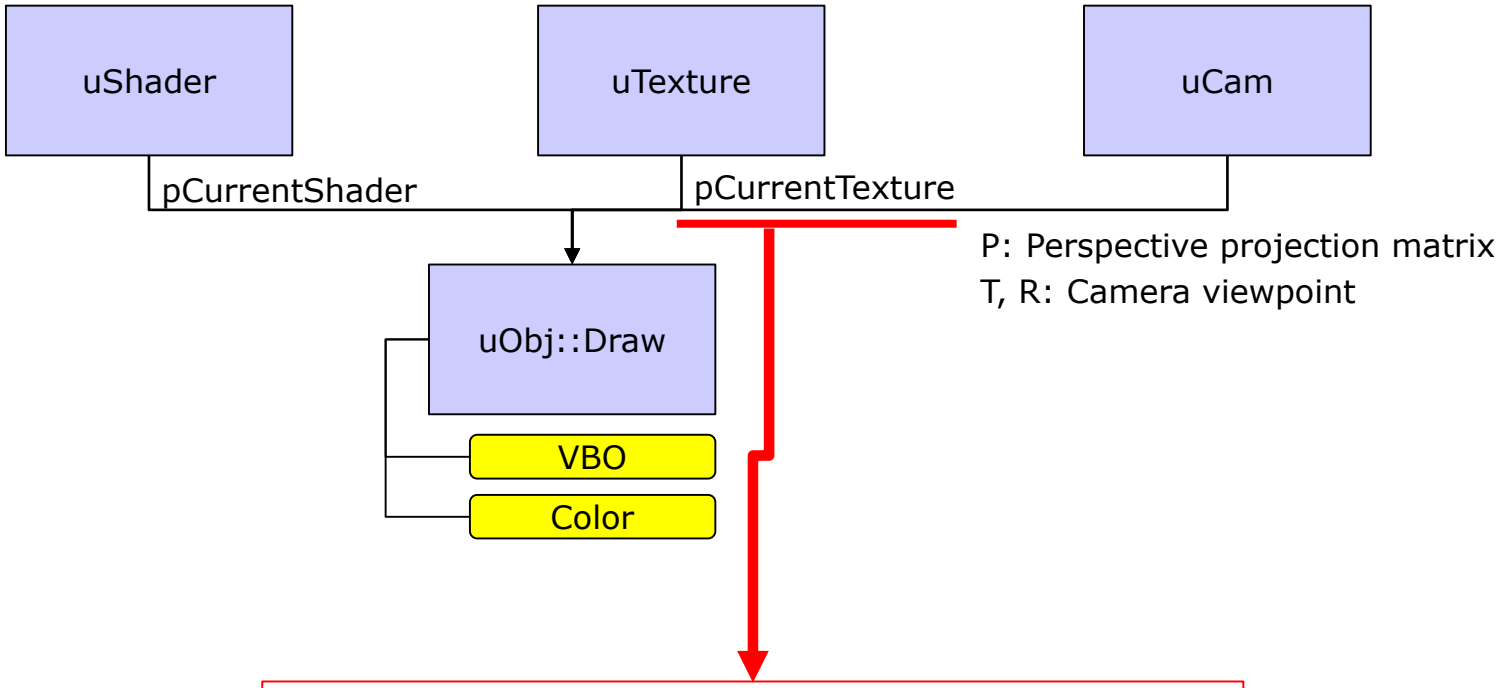
# Step 2: Draw() in a New Structure
# step 2-3: Vertex buffer(VBO)




pp. 26,
Lecture 9

```
// call vertex buffer
glBindBuffer(GL_ARRAY_BUFFER, vs);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0,3, GL_FLOAT,FALSE,sizeof(uVertex),(void*)0);    // vertex,v
glEnableVertexAttribArray(1);
glVertexAttribPointer(1,3, GL_FLOAT,FALSE,sizeof(uVertex),(void*)12);   // normal,n
glEnableVertexAttribArray(2);
glVertexAttribPointer(2,2, GL_FLOAT,FALSE,sizeof(uVertex),(void*)24);   // texture uv

// call face buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, fs);
```

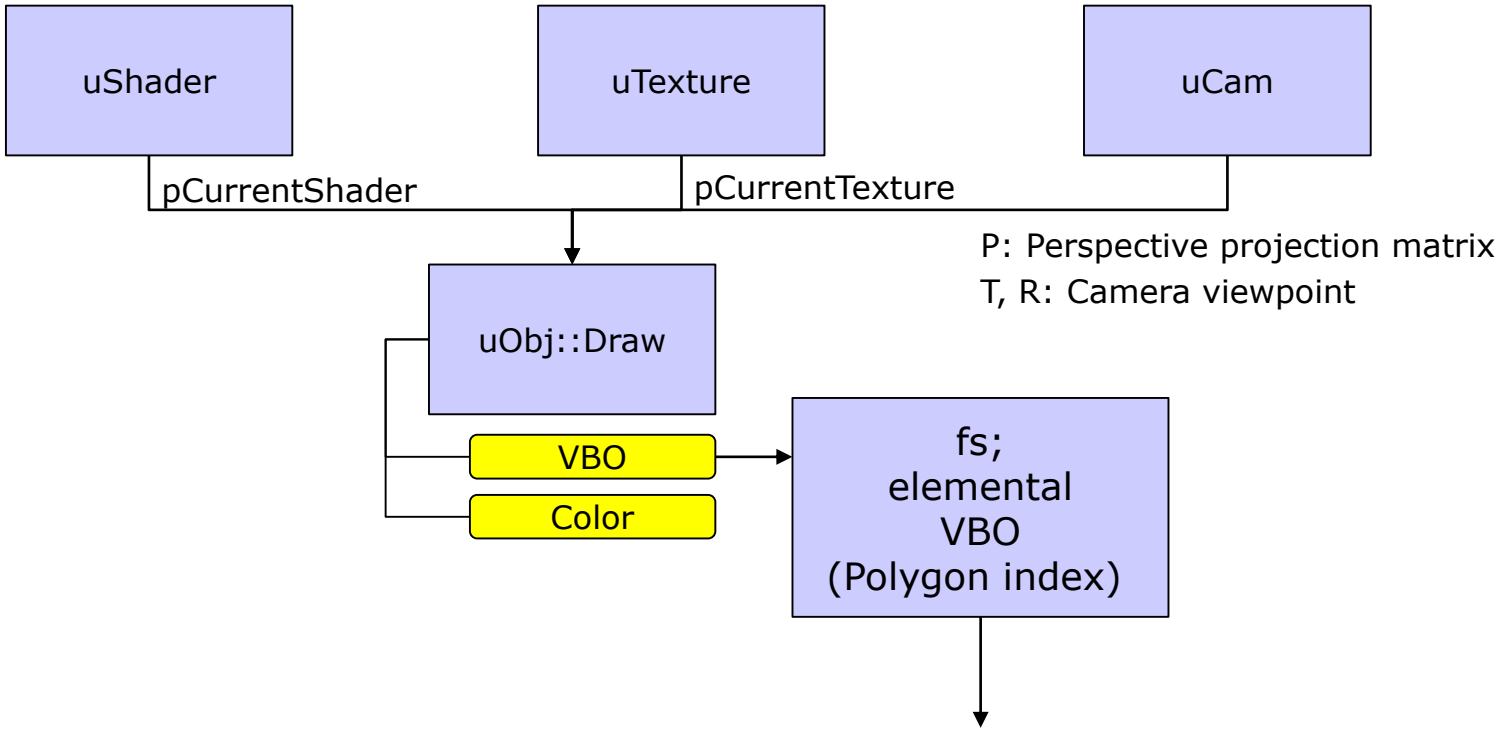
# Step 2: Draw() in a New Structure
# step 2-4: Texture mapping




```
// call texture
if (pTX)
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D,pTX->tid);
}
```



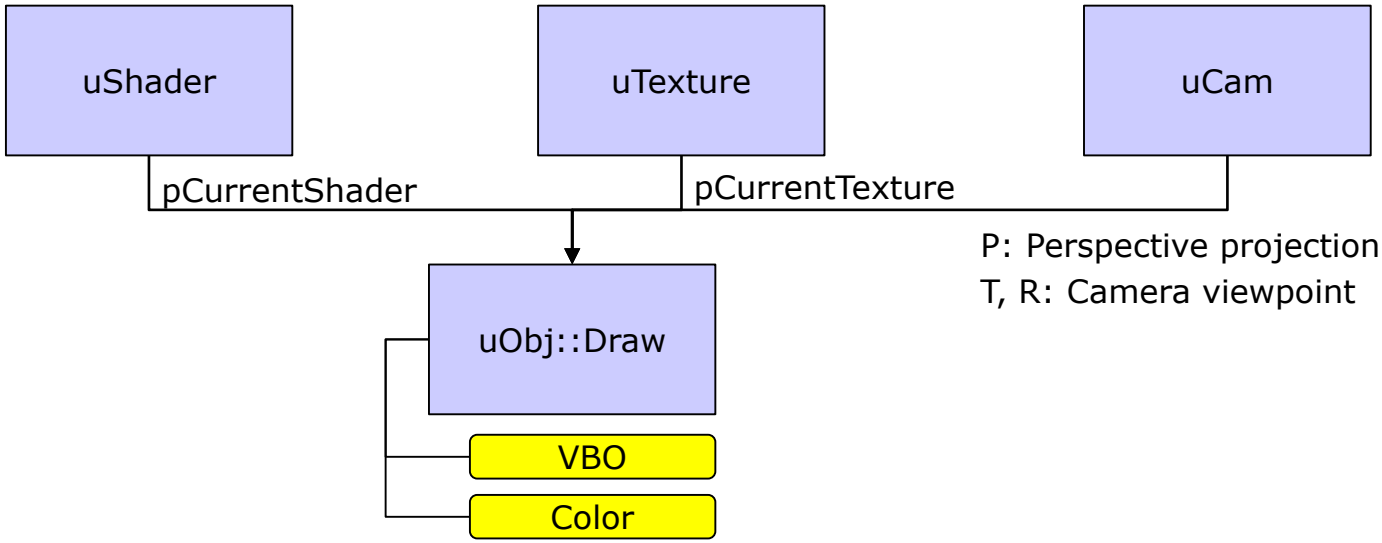
# Step 2: Draw() in a New Structure
# step 2-5: Draw Element




```
glDrawElements(GL_TRIANGLES, 3*nPoly, GL_UNSIGNED_SHORT, (void*)0);
```



# Step 2: Draw() in a New Structure
# step 2-6: Finish Drawing




```
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(2);

glBindBuffer(GL_ARRAY_BUFFER,0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

Close VBO handle

```
if (pTX)
{
    glDisable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D,0);
}
```

Close Texture buffer handle



# Step3 Drawing Multiple Object in uWnd

```
void uWnd::Draw()
{
    glClearColor(info.r,info.g,info.b,info.a);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // draw objects
    for (int i=0;i<objs.GetSize();i++)
    objs[i]->Draw();

    glFinish();
}
```

```
public:
        vArray<uObj*,uObj*>          objs;
```

```
void uObj::Draw()
{
    // camera coordinate setting
    hMat h   = pCam->T*pCam->R;
    h    = h*Hg*H*Hp;

    glUniformMatrix4fv(pCurrentShader->screen, 1, 0, pCam->P.v);
    glUniformMatrix4fv(pCurrentShader->model, 1, 0, h.v);
```

# Step 4: Object Transform
## ex) uGL-27-Complete-object-transform

- We used to modify H matrix of an object
  - See uGL-26-Complete-GL objects

```
void uWnd::Run()
{
    // Where box?
    uObj *p = objs[0];

    p->q.y+=1;
    p->H        = p->H.RotY(p->q.y)*p->H.Trans(-5,0,0);
    Redraw();
}
```

- We will uses uObj::Trans and uObj::Rot
  - Extra information is required for **storing position and rotation**

uObj

```
// Transform
hMat    Hg;
hMat    H;

struct _coord
{
    uVector o;
    uVector q;
} coord;
```



# uObj::Trans and uObj::Rot

```
void uObj::Trans(float x,float y,float z)
{
    coord.o = uVector(x,y,z);
    H.v[12] = x;
    H.v[13] = y;
    H.v[14] = z;
}
```

```
void uObj::Rot(float x,float y,float z)
{
    coord.q = uVector(x,y,z);
    hMat t  = H.RotZ(z)*H.RotY(y)*H.RotX(x);

    // copy 0 1 2, 4,5,6, 8,9,10,
    H.v[0] = t.v[0];    H.v[1] = t.v[1];    H.v[2] = t.v[2];
    H.v[4] = t.v[4];    H.v[5] = t.v[5];    H.v[6] = t.v[6];
    H.v[8] = t.v[8];    H.v[9] = t.v[9];    H.v[10]= t.v[10];
}
```

- Sometimes, **uObj::Trans(x,y,z)** is boring,
- How about using **uObj::Trans(uVector)**?
  - Use C++ overloading.

```
void uObj::Trans(uVector v)
{
    Trans(v.x,v.y,v.z);
}
```

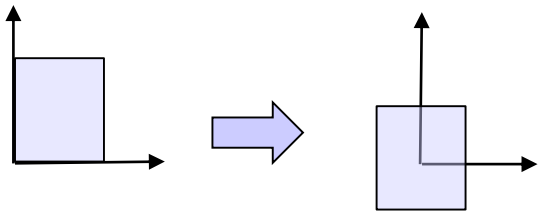
```
void uObj::Rot(uVector v)
{
    Rot(v.x,v.y,v.z);
}
```

**Overloading in C++ is very helpful.**



# Step 5: Pivotal Rotation (Hp)

## ex) uGL-28-Complete-object-transform-pivot



$$H = H_{Trans} H_{Rot} H_{-Trans}$$

Pivotal rotation pp.28 in lecture 5

- New function, uObj::Pivot requires : $H_p = H_{Trans}$

```
void uObj::Pivot(float x,float y,float z)
{
    Hp  = Hp.Trans(-x,-y,-z);
}
```

- **Changes in Draw()**

```
void uObj::Draw()
{
    // camera coordinate setting
    hMat h  = pCam->T*pCam->R;
    h   = h*H*Hp;

    glUniformMatrix4fv(pCurrentShader->screen, 1, 0, pCam->P.v);
    glUniformMatrix4fv(pCurrentShader->model, 1, 0, h.v);
```



# Comparison Pivot

uGL-27-Complete-object-transform

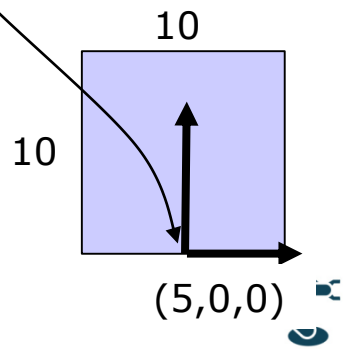
```
void uWnd::Loading()
{
    SetBK(RGB(255,255,255));
    m_cam.T = m_cam.T.Trans(0,0,-10);

    // shader and texture
    pCurrentShader  = CreateShader("Ph
    pCurrentTexture = CreateTexture("h

    // object creation
    uObj *p = CreateObj();
    p->MakeBox(10,10,0.1);
    p->pTX       = pCurrentTexture;
    p->diffuse   = uColor(1,1,1);
    p->ambient   = uColor(0.1,0.1,0.1);
    p->specular  = uColor(0,0,0);
    p->Update();
}
```

```
void uWnd::Run()
{
    // Where box?
    uObj *p = objs[0];

    // Previous code
    p->q.y+=1;
    p->H         = p->H.RotY(p->q.y)*p->H.Trans(-5,0,0);

    Redraw();
}
```

uGL-28-Complete-object-transform-pivot

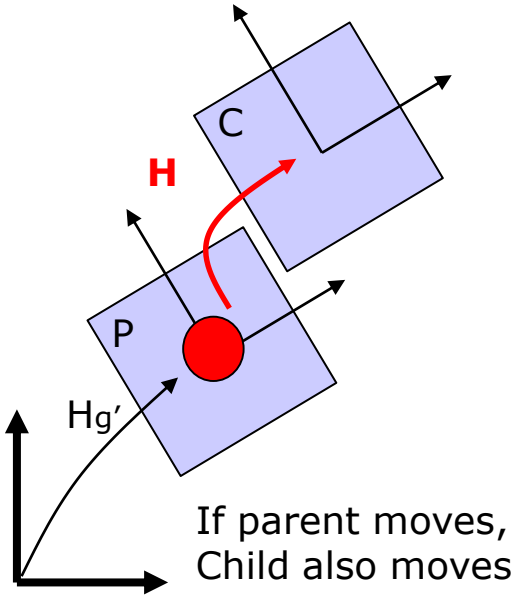
```
void uWnd::Loading()
{
    SetBK(RGB(255,255,255));
    m_cam.T = m_cam.T.Trans(0,0,-20);

    // shader and texture
    pCurrentShader  = CreateShader("Ph
    pCurrentTexture = CreateTexture("h

    // object creation
    uObj *p = CreateObj();
    p->MakeBox(10,10,0.1);
    p->pTX       = pCurrentTexture;
    p->diffuse   = uColor(1,1,1);
    p->ambient   = uColor(0.1,0.1,0.1);
    p->specular  = uColor(0,0,0);
    p->Pivot(5,0,0);

    p->Update();
}
```

```
void uWnd::Run()
{
    // Where box?
    uObj *p = objs[0];

    p->coord.q.y++;
    p->Rot(p->coord.q);

    Redraw();
}
```



# Step 6: Hierarchical Aspect of Objects

- Think Robot Manipulator (pp. 18 in lecture 5)

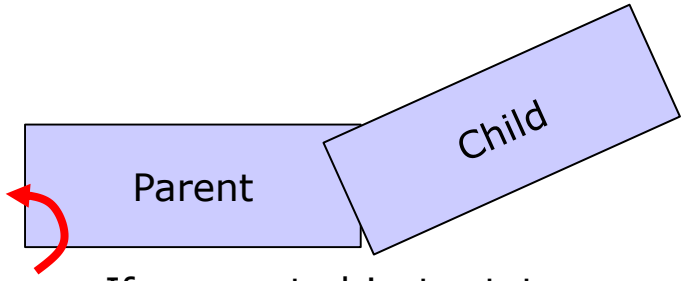



If parent moves,
Child also moves

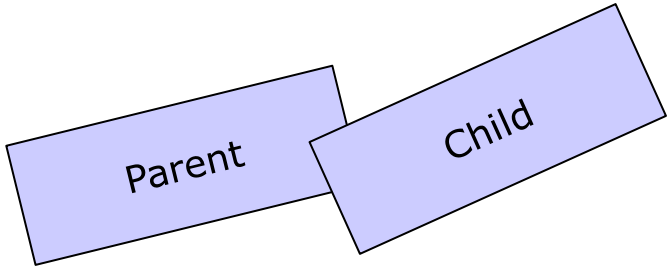
$$H_g = H_g'$$

$$P_{current} = H_g' P$$

$$C_{current} = H_g' HC$$



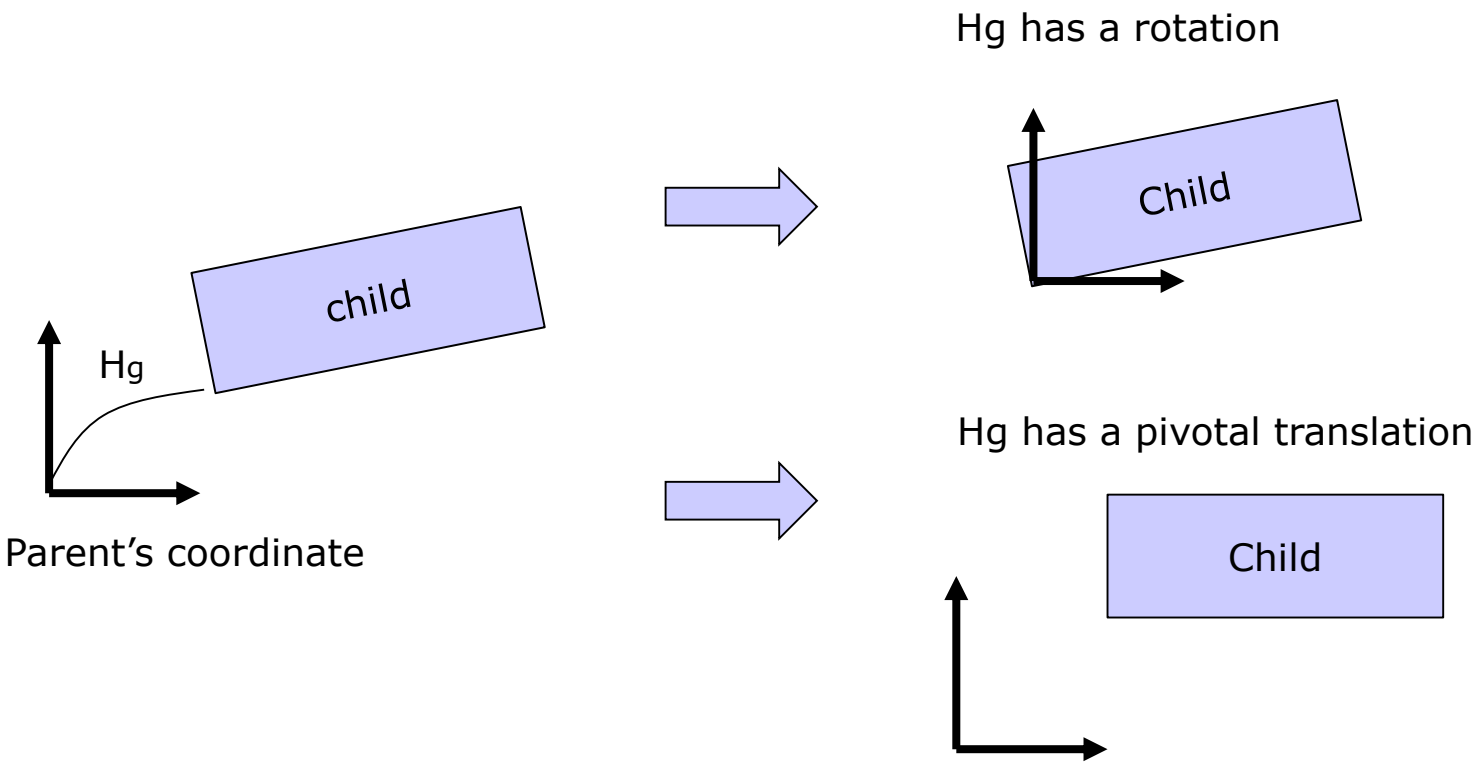

If a parent object rotates,




fail

# All Complex Coordinates in uObj

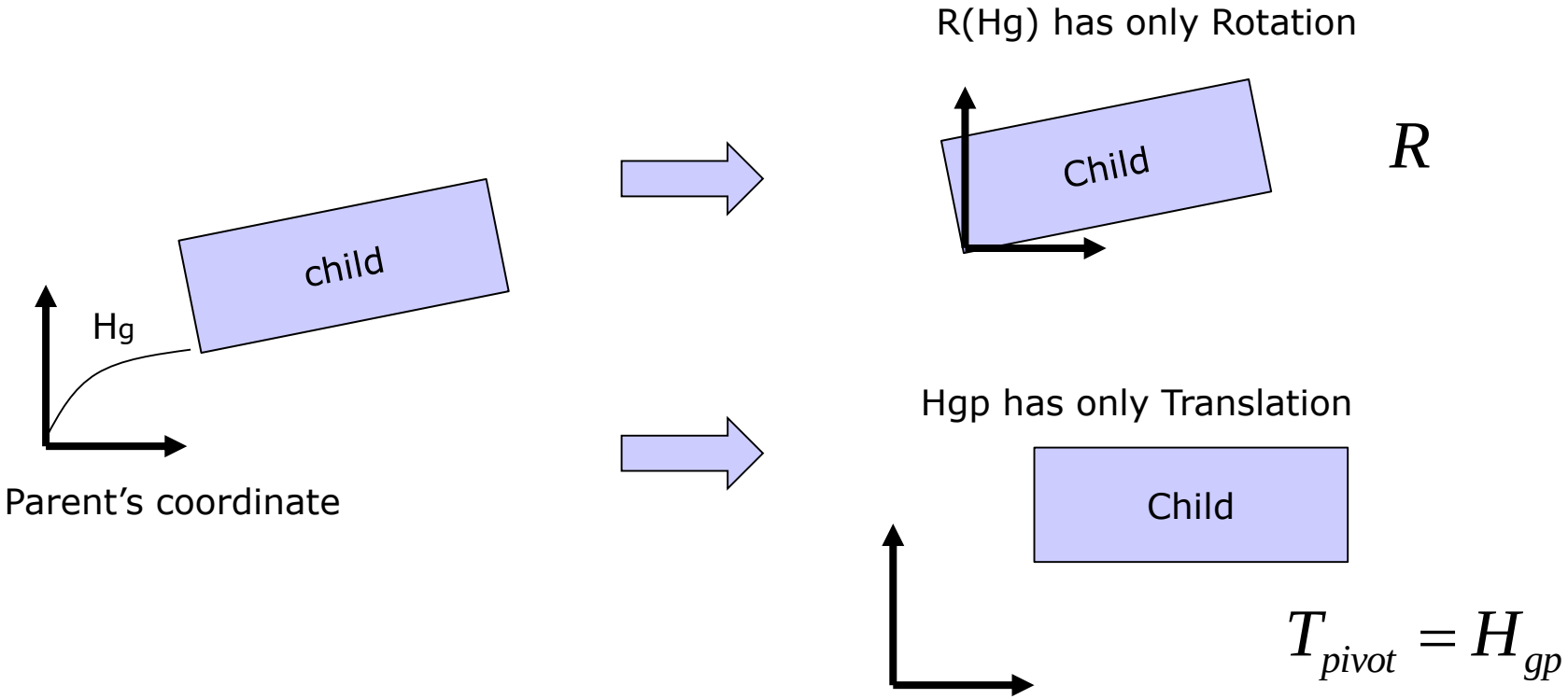
## uGL-31-Complete-object-Hierarchy-Final




- Define **Hg is a combination of Translation and Rotation**

With respect to Parent's coordinate

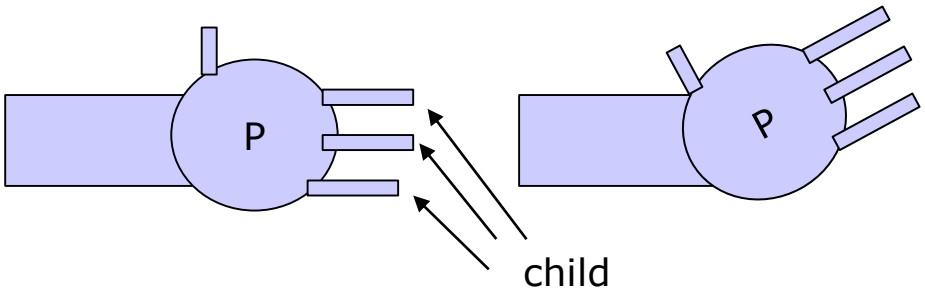
# For convenient usages, Define Pivotal Translation, Hgp, from Parent




$$H_g = RT_{pivot} = R \bullet H_{gp}$$

# uObj in Hierarchical Structure

- uObj has children objects




```
class uObj
{
public:
     uObj();
     ~uObj();
public:
     ...

     // child object
     vArray<uObj*,uObj*> child;

     ...
};
```

- uObj::Draw also draws children

```
void uObj::Draw()
{
     ...

     for (int i=0;i<child.GetSize();i++)
     child[i]->Draw();
}
```



# uObj::Rot in Hierarchical Structure

```
void uObj::Rot(float x,float y,float z)
{
    coord.q = uVector(x,y,z);
    hMat r  = H.RotZ(z)*H.RotY(y)*H.RotX(x);

    // copy 0 1 2, 4,5,6, 8,9,10,
    H.v[0] = r.v[0];    H.v[1] = r.v[1];    H.v[2] = r.v[2];
    H.v[4] = r.v[4];    H.v[5] = r.v[5];    H.v[6] = r.v[6];
    H.v[8] = r.v[8];    H.v[9] = r.v[9];    H.v[10]= r.v[10];

    hMat h;
    h   = Hg*H;

    for (int i=0;i<child.GetSize();i++)
    {
        child[i]->Hg    = h*child[i]->Hgp;
        child[i]->Rot(child[i]->coord.q);
    }
}
```

$H^p$

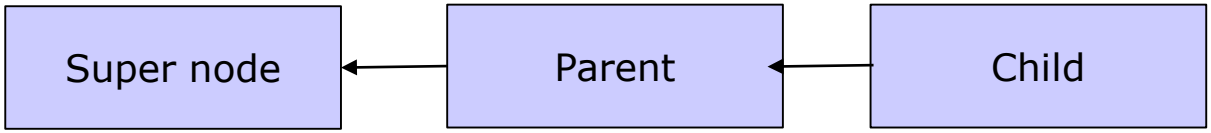
$$H_g^c = \left( H_g^p H^p \right) H_{gp}^c$$

**Why child[i]→Rot is called here?**

It is a tactical method that rotation is propagated through every children in uObj



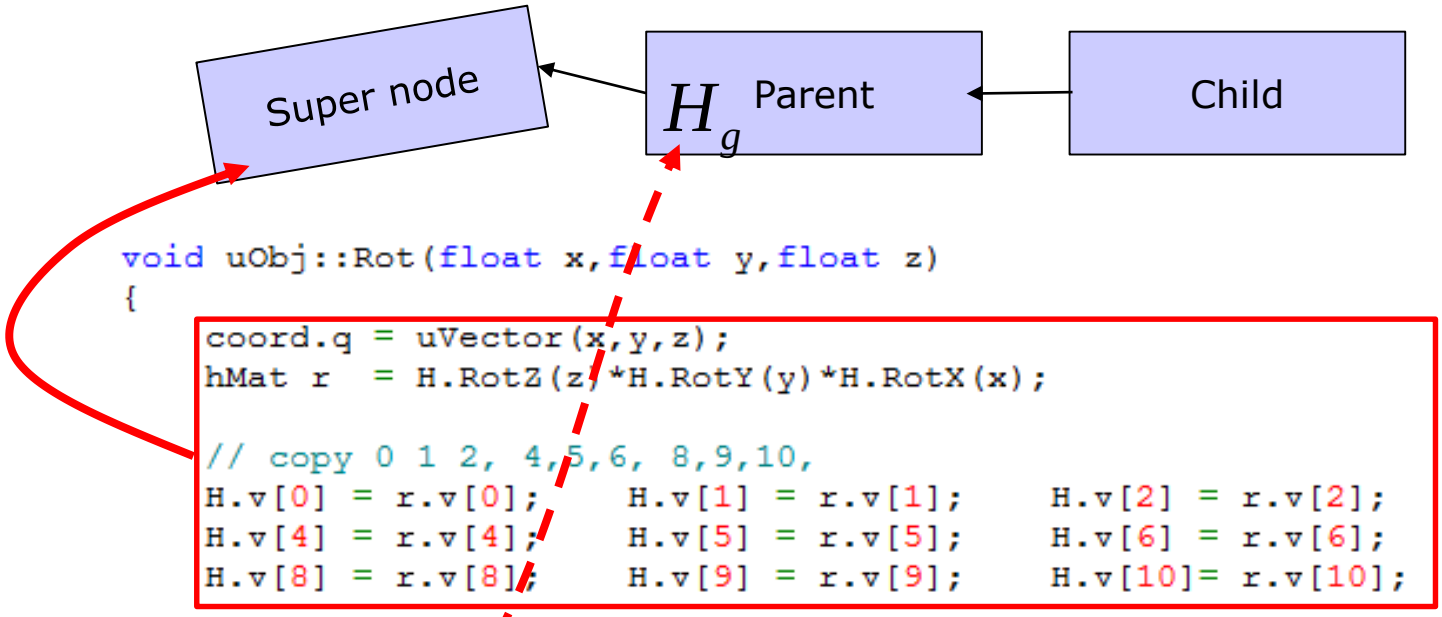
# Extra Tip for Understanding Rot in Hierarchical Structure




```
void uObj::Rot(float x,float y,float z)
{
    coord.q = uVector(x,y,z);
    hMat r  = H.RotZ(z)*H.RotY(y)*H.RotX(x);

    // copy 0 1 2, 4,5,6, 8,9,10,
    H.v[0] = r.v[0];    H.v[1] = r.v[1];    H.v[2] = r.v[2];
    H.v[4] = r.v[4];    H.v[5] = r.v[5];    H.v[6] = r.v[6];
    H.v[8] = r.v[8];    H.v[9] = r.v[9];    H.v[10]= r.v[10];

    hMat h;
    h   = Hg*H;

    for (int i=0;i<child.GetSize();i++)
    {
        child[i]->Hg    = h*child[i]->Hgp;
        child[i]->Rot(child[i]->coord.q);
    }
}
```



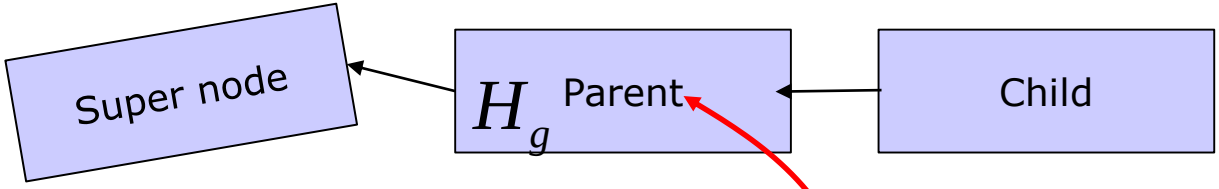
# Extra Tip for Understanding Rot in Hierarchical Structure




```
void uObj::Rot(float x,float y,float z)
{
    coord.q = uVector(x,y,z);
    hMat r  = H.RotZ(z)*H.RotY(y)*H.RotX(x);

    // copy 0 1 2, 4,5,6, 8,9,10,
    H.v[0] = r.v[0];    H.v[1] = r.v[1];    H.v[2] = r.v[2];
    H.v[4] = r.v[4];    H.v[5] = r.v[5];    H.v[6] = r.v[6];
    H.v[8] = r.v[8];    H.v[9] = r.v[9];    H.v[10]= r.v[10];

    hMat h;
    h   = Hg*H;

    for (int i=0;i<child.GetSize();i++)
    {
        child[i]->Hg    = h*child[i]->Hgp;
        child[i]->Rot(child[i]->coord.q);
    }
}
```



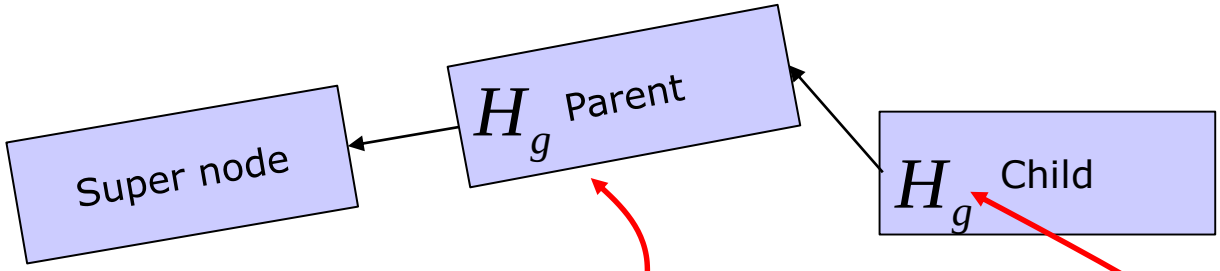
# Extra Tip for Understanding Rot in Hierarchical Structure




```
void uObj::Rot(float x,float y,float z)
{
    coord.q = uVector(x,y,z);
    hMat r  = H.RotZ(z)*H.RotY(y)*H.RotX(x);

    // copy 0 1 2, 4,5,6, 8,9,10,
    H.v[0] = r.v[0];    H.v[1] = r.v[1];    H.v[2] = r.v[2];
    H.v[4] = r.v[4];    H.v[5] = r.v[5];    H.v[6] = r.v[6];
    H.v[8] = r.v[8];    H.v[9] = r.v[9];    H.v[10]= r.v[10];

    hMat h;
    h   = Hg*H;

    for (int i=0;i<child.GetSize();i++)
    {
        child[i]->Hg    = h*child[i]->Hgp;
        child[i]->Rot(child[i]->coord.q);
    }
}
```



# Extra Tip for Understanding Rot in Hierarchical Structure

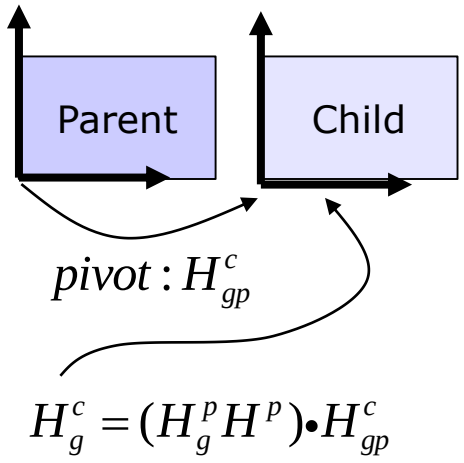


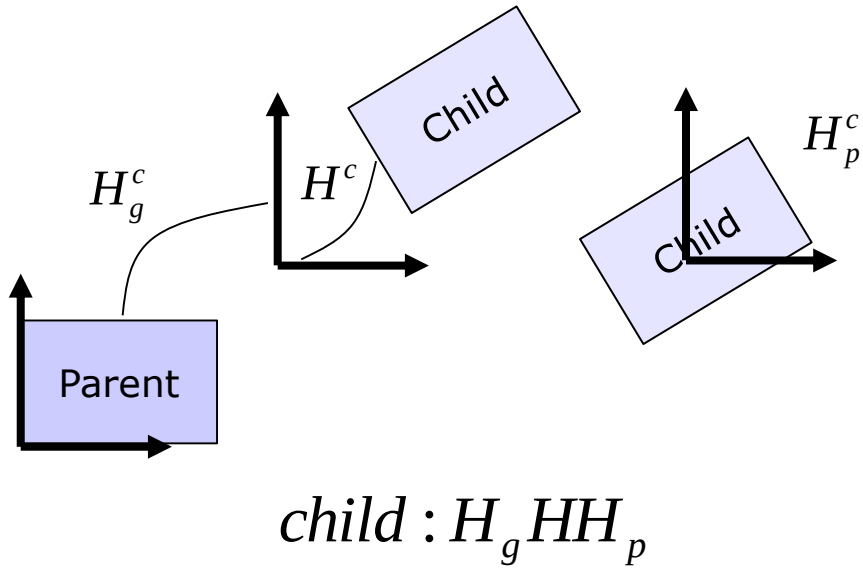

```
void uObj::Rot(float x,float y,float z)
{
    coord.q = uVector(x,y,z);
    hMat r  = H.RotZ(z)*H.RotY(y)*H.RotX(x);

    // copy 0 1 2, 4,5,6, 8,9,10,
    H.v[0] = r.v[0];    H.v[1] = r.v[1];    H.v[2] = r.v[2];
    H.v[4] = r.v[4];    H.v[5] = r.v[5];    H.v[6] = r.v[6];
    H.v[8] = r.v[8];    H.v[9] = r.v[9];    H.v[10]= r.v[10];

    hMat h;
    h   = Hg*H;

    for (int i=0;i<child.GetSize();i++)
    {
        child[i]->Hg    = h*child[i]->Hgp;
        child[i]->Rot(child[i]->coord.q);
    }
}
```



# uObj::Draw
# Parent Transform modifies Child's Hg

```
void uObj::Draw()
{
    // camera coordinate setting
    hMat h   = pCam->T*pCam->R;
    h    = h*Hg*H*Hp;

    glUniformMatrix4fv(pCurrentShader->screen, 1, 0, pCam->P.v);
    glUniformMatrix4fv(pCurrentShader->model, 1, 0, h.v);
```



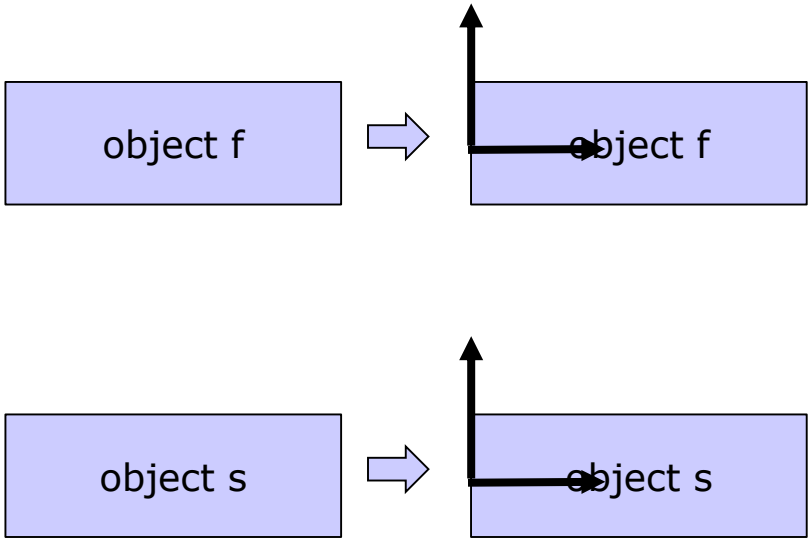






# Ex) Final Integration of 3D Graphics in OpenGL

## uGL-31-Complete-object-Hierarchy-Final

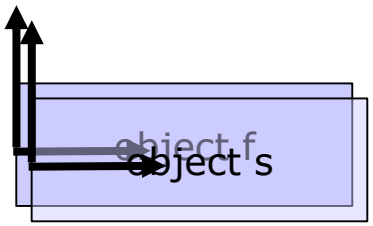
```
void uWnd::Loading()
{
    ...
    // object creation
    uObj *f = CreateObj();
    f->MakeBox(10,5,5);
    f->pTX      = pCurrentTexture;
    f->diffuse  = uColor(1,1,1);
    f->ambient  = uColor(0.1,0.1,0.1);
    f->specular = uColor(0,0,0);
    f->Pivot(0,2.5,2.5);
    f->Update();

    uObj *s = CreateObj();
    s->MakeBox(10,5,5);
    s->pTX      = pCurrentTexture;
    s->diffuse  = uColor(1,1,1);
    s->ambient  = uColor(0.1,0.1,0.1);
    s->specular = uColor(0,0,0);
    s->Pivot(0,2.5,2.5);
    s->Update();

    // s is a child of f.
    f->Add(s);

    // s is moved by length of f
    s->Hgp  = s->Hgp.Trans(10,0,0);

    // f and s move independently
    f->Trans(0,10,0);
    s->Rot(0,0,30);
}
```

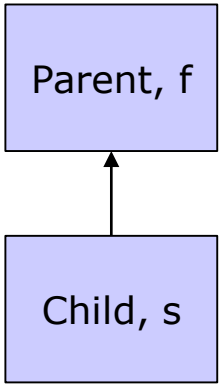


# Ex) Final Integration of 3D Graphics in OpenGL

## uGL-31-Complete-object-Hierarchy-Final

```
void uWnd::Loading()
{
    ...
    // object creation
    uObj *f = CreateObj();
    f->MakeBox(10,5,5);
    f->pTX      = pCurrentTexture;
    f->diffuse  = uColor(1,1,1);
    f->ambient  = uColor(0.1,0.1,0.1);
    f->specular = uColor(0,0,0);
    f->Pivot(0,2.5,2.5);
    f->Update();

    uObj *s = CreateObj();
    s->MakeBox(10,5,5);
    s->pTX      = pCurrentTexture;
    s->diffuse  = uColor(1,1,1);
    s->ambient  = uColor(0.1,0.1,0.1);
    s->specular = uColor(0,0,0);
    s->Pivot(0,2.5,2.5);
    s->Update();

    // s is a child of f.
    f->Add(s);

    // s is moved by length of f
    s->Hgp  = s->Hgp.Trans(10,0,0);

    // f and s move independently
    f->Trans(0,10,0);
    s->Rot(0,0,30);
}
```



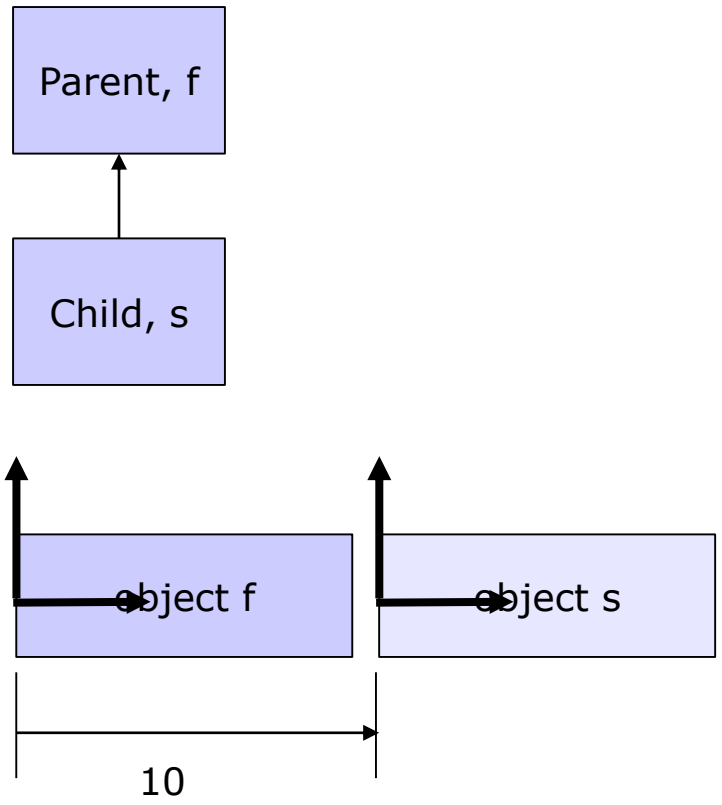






# Ex) Final Integration of 3D Graphics in OpenGL

## uGL-31-Complete-object-Hierarchy-Final

```
void uWnd::Loading()
{
    ...
    // object creation
    uObj *f = CreateObj();
    f->MakeBox(10,5,5);
    f->pTX      = pCurrentTexture;
    f->diffuse  = uColor(1,1,1);
    f->ambient  = uColor(0.1,0.1,0.1);
    f->specular = uColor(0,0,0);
    f->Pivot(0,2.5,2.5);
    f->Update();

    uObj *s = CreateObj();
    s->MakeBox(10,5,5);
    s->pTX      = pCurrentTexture;
    s->diffuse  = uColor(1,1,1);
    s->ambient  = uColor(0.1,0.1,0.1);
    s->specular = uColor(0,0,0);
    s->Pivot(0,2.5,2.5);
    s->Update();

    // s is a child of f.
    f->Add(s);

    // s is moved by length of f
    s->Hgp  = s->Hgp.Trans(10,0,0);

    // f and s move independently
    f->Trans(0,10,0);
    s->Rot(0,0,30);
}
```

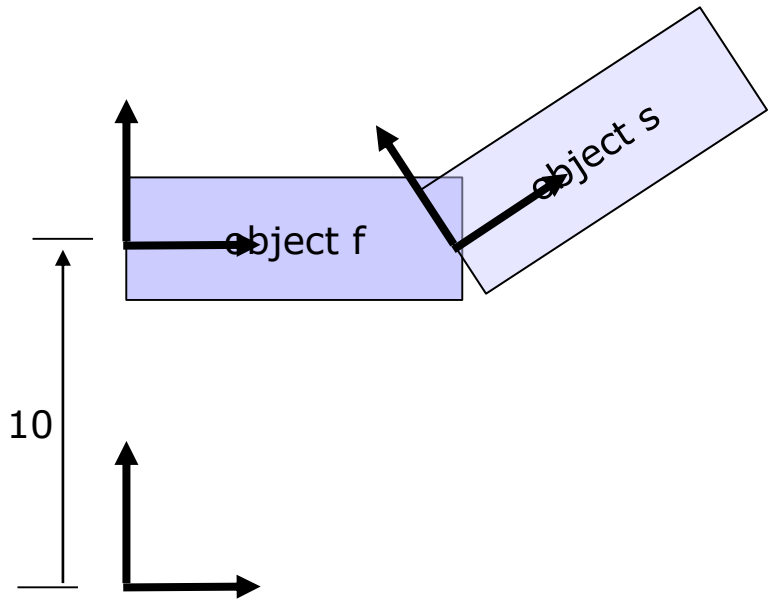





# Ex) Final Integration of 3D Graphics in OpenGL

## uGL-31-Complete-object-Hierarchy-Final

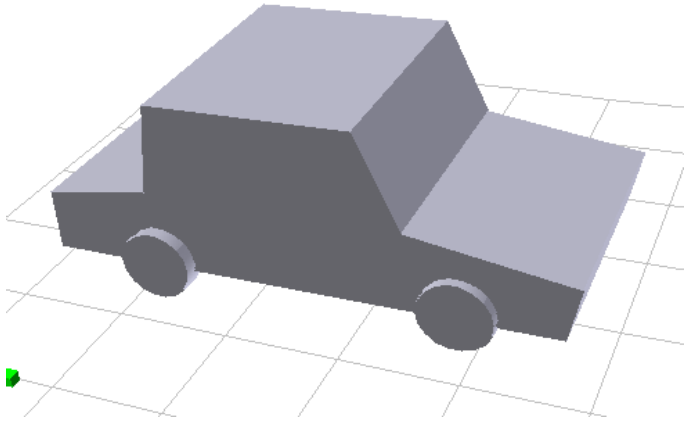
```
void uWnd::Loading()
{
    ...
    // object creation
    uObj *f = CreateObj();
    f->MakeBox(10,5,5);
    f->pTX      = pCurrentTexture;
    f->diffuse  = uColor(1,1,1);
    f->ambient  = uColor(0.1,0.1,0.1);
    f->specular = uColor(0,0,0);
    f->Pivot(0,2.5,2.5);
    f->Update();

    uObj *s = CreateObj();
    s->MakeBox(10,5,5);
    s->pTX      = pCurrentTexture;
    s->diffuse  = uColor(1,1,1);
    s->ambient  = uColor(0.1,0.1,0.1);
    s->specular = uColor(0,0,0);
    s->Pivot(0,2.5,2.5);
    s->Update();

    // s is a child of f.
    f->Add(s);

    // s is moved by length of f
    s->Hgp  = s->Hgp.Trans(10,0,0);

    // f and s move independently
    f->Trans(0,10,0);
    s->Rot(0,0,30);
}
```



# You have learned and finished the Cores of 3D graphics

# 2 Test for Final Graphics Engine

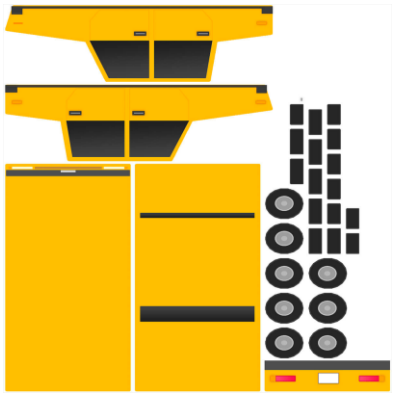
# Ex) uGL-36-Complete-Car step 1: Allocation



Car.obj

- Position 2194
- UV 2194
- Face 324

```
// car creation
uObj *p = CreateObj();
p->Alloc(2194,324);
```



Car_07.png



# Ex) uGL-36-Complete-Car
# step 2: Position and UV Vector

```
// load vertices
uVector sum;
for (i=0;i<2194;i++)
{
    p->pVer[i].v.x  = carv[3*i];
    p->pVer[i].v.y  = carv[3*i+1];
    p->pVer[i].v.z  = carv[3*i+2];

    sum = sum+p->pVer[i].v;

    p->pVer[i].tx.u = cartx[2*i];
    p->pVer[i].tx.v = cartx[2*i+1];
}
sum = sum*(1./2194);
```

Position

Get Center for pivot

Texture Mapping (UV)



# Ex) uGL-36-Complete-Car step 3: Normal Vector

```
// load faces
uVector vf,vs,n;
for (i=0;i<324;i++)
{
    unsigned short f,s,t;
    f = carface[3*i]-1;;
    s = carface[3*i+1]-1;
    t = carface[3*i+2]-1;

    p->pPoly[i].f  = f;
    p->pPoly[i].s  = s;
    p->pPoly[i].t  = t;

    vf = (p->pVer[t].v-p->pVer[s].v);
    vs = (p->pVer[f].v-p->pVer[s].v);
    n = vf*vs;
    n = n.Unit();
    p->pVer[f].n = n;
    p->pVer[s].n = n;
    p->pVer[t].n = n;
}
```

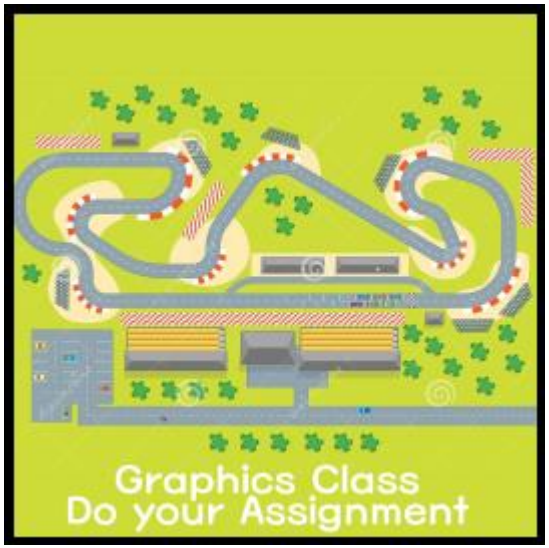
Polygon indexing

Get Normal of polygon

Semi Flat shading

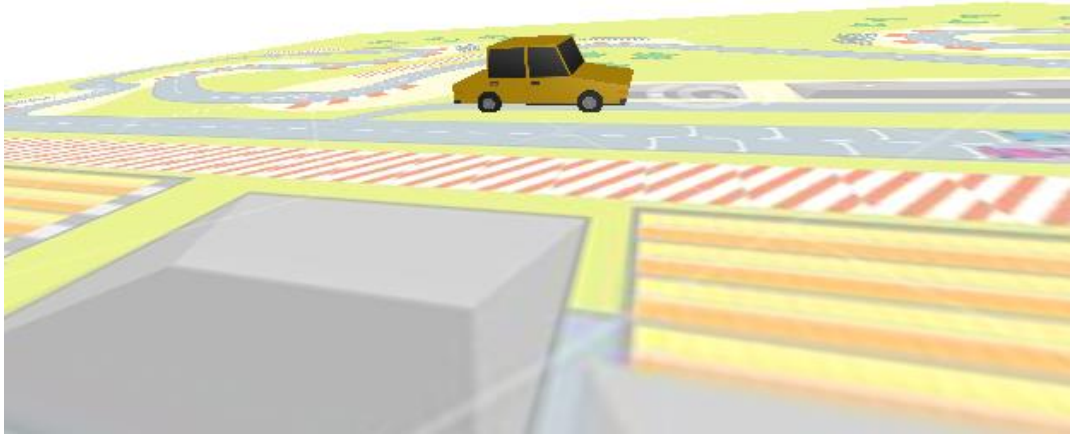


# Ex) uGL-36-Complete-Car
# step 4: Ground with Texture mapping

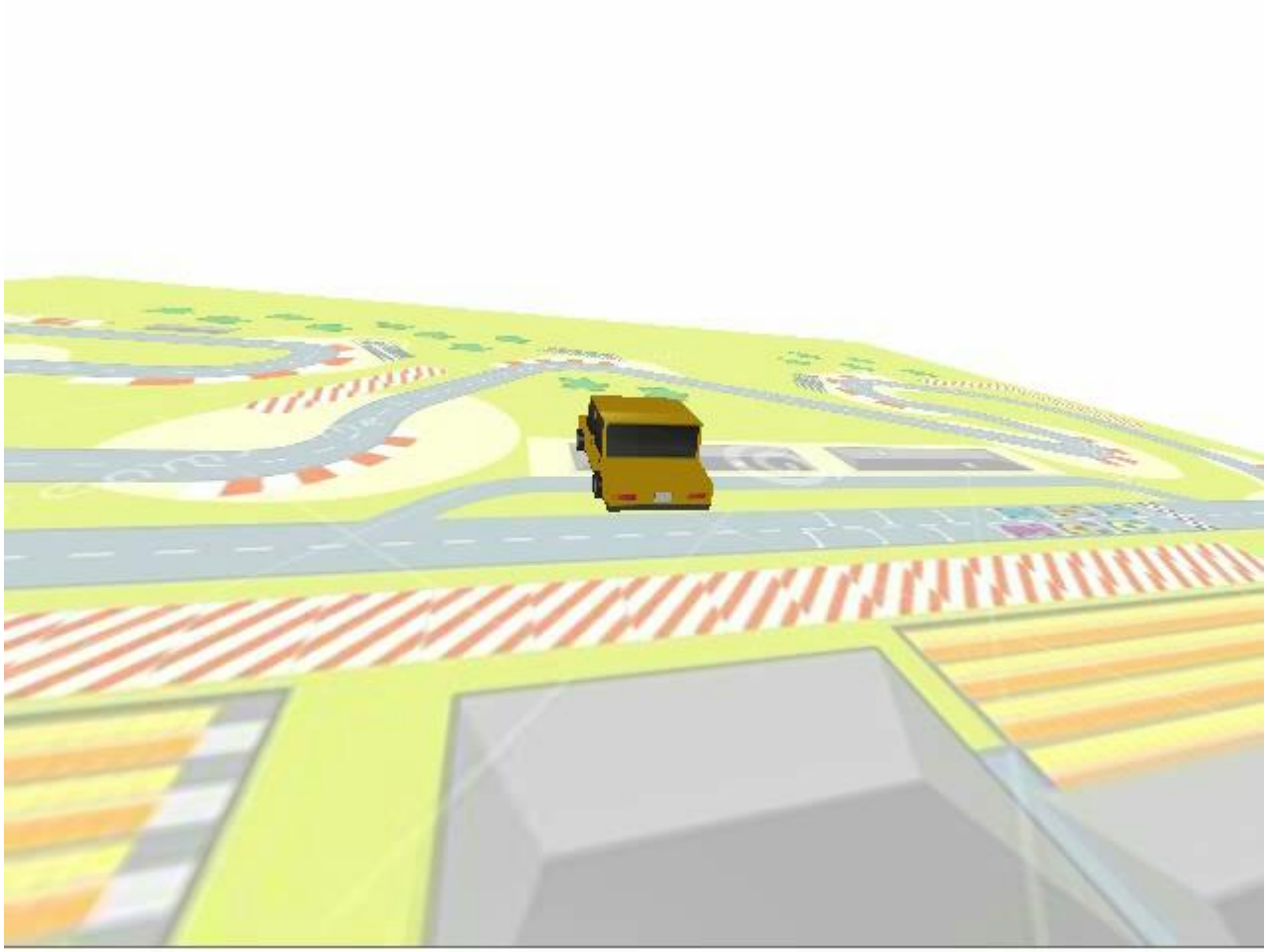



Road.png

- MakeBox (3000,3000,1)



# Ex) uGL-37-Complete-Car-bykey-Ans



# Ex) uGL-38-Complete-Car-byMouse-Ans

- Right button for moving a car

# Ex) uGL-38-Complete-Car-byMouse-Ans

```
// Projection matrix
float n=1;
float f=65535;
float angle      = 90;
float aspect     = 640./480.;
float ct   = 1./tan(RAD(angle)/2);
```

- OpenGL uses
  - n=1 and f=65535
  - Width=640
  - Height=480

- Ex) uWnd-49-MC-Inv
  - Refer to pp.28. in lecture 6

- Key input Message **bypassing** in testview.cpp

```
void CtestView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    gl.SendMessage(WM_KEYDOWN,nChar,0);
    CView::OnKeyDown(nChar, nRepCnt, nFlags);
}
```



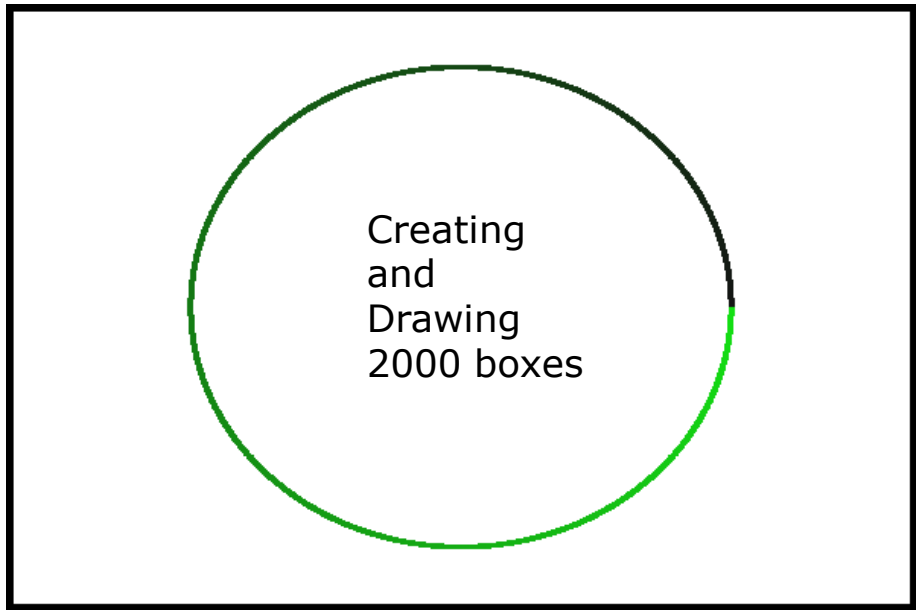
# 3 Multi Objects in Graphics

# Multiple Objects Creation

## Ex) uGL-32-Complete-MultiObject-A

```
void uWnd::Loading()
{
    SetBK(RGB(255,255,255));
    m_cam.T = m_cam.T.Trans(0,0,-100);

    // shader and texture
    pCurrentShader  = CreateShader("PhongTe:

    for (int i=0;i<2000;i++)
    {
        uObj *p      = CreateObj();
        p->MakeBox(1,1,1);
        p->pTX       = NULL;

        float r=0,g=0,b=0;
        g = i;
        g = g/2000.;

        p->diffuse  = uColor(2*r,2*g,2*b);
        p->ambient  = uColor(0.2,0.2,0.2);
        p->specular = uColor(1,1,1);

        p->Pivot(0.5,0.5,0.5);

        float R= 80;
        float dq = (360./2000);
        float x = R*cos(RAD(dq*i));
        float y = R*sin(RAD(dq*i));
        p->Trans(x,y,0);
        p->Update();
    }
}
```

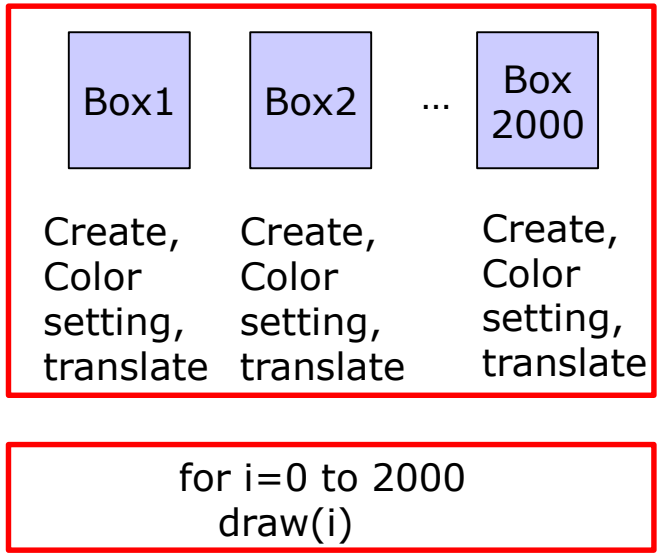
-Create 2000 boxes.
-Green color is varying.
-Translate each box along a circular path



# Think Everything in a Different Way

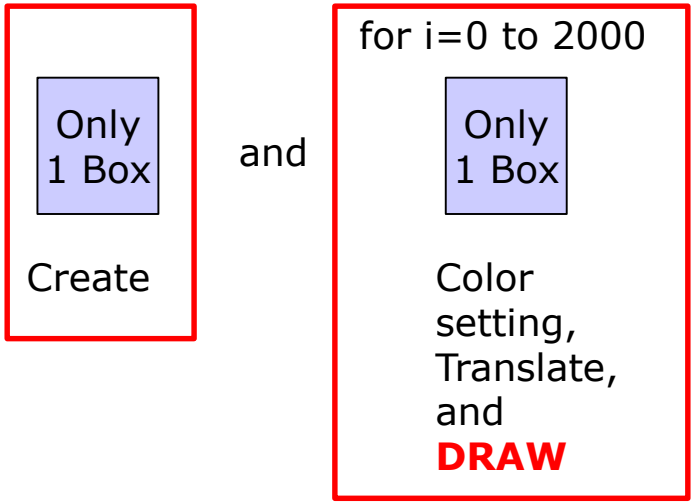
## Ex) uGL-32-Complete-MultiObject-B

Ex) uGL-32-Complete-MultiObject-A




Vs.

Ex) uGL-32-Complete-MultiObject-B




- What will happen?



# Differences in Script in Loading

Ex) uGL-32-Complete-MultiObject-A

```
void uWnd::Loading()
{
    SetBK(RGB(255,255,255));
    m_cam.T = m_cam.T.Trans(0,0,-100);

    // shader and texture
    pCurrentShader  = CreateShader("PhongTe:

    for (int i=0;i<2000;i++)
    {
        uObj *p      = CreateObj();
        p->MakeBox(1,1,1);
        p->pTX       = NULL;

        float r=0,g=0,b=0;
        g = i;
        g = g/2000.;

        p->diffuse  = uColor(2*r,2*g,2*b);
        p->ambient  = uColor(0.2,0.2,0.2);
        p->specular = uColor(1,1,1);

        p->Pivot(0.5,0.5,0.5);

        float R= 80;
        float dq = (360./2000);
        float x = R*cos(RAD(dq*i));
        float y = R*sin(RAD(dq*i));
        p->Trans(x,y,0);
        p->Update();
    }
}
```

Ex) uGL-32-Complete-MultiObject-B

```
void uWnd::Loading()
{
    SetBK(RGB(255,255,255));
    m_cam.T = m_cam.T.Trans(0,0,-100);

    // shader and texture
    pCurrentShader  = CreateShader("Pho:

    // box
    uObj *p      = CreateObj();
    p->MakeBox(1,1,1);
    p->pTX       = NULL;

    p->diffuse  = uColor(0,0,0);
    p->ambient  = uColor(0.2,0.2,0.2);
    p->specular = uColor(1,1,1);

    p->Pivot(0.5,0.5,0.5);
    p->Update();
}
```



# Differences in Script in Draw

Ex) uGL-32-Complete-MultiObject-A

```
void uWnd::Draw()
{
    glClearColor(info.r,info.g,info.b,info.a);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // draw objects
    for (int i=0;i<objs.GetSize();i++)
    objs[i]->Draw();

    glFinish();
}
```

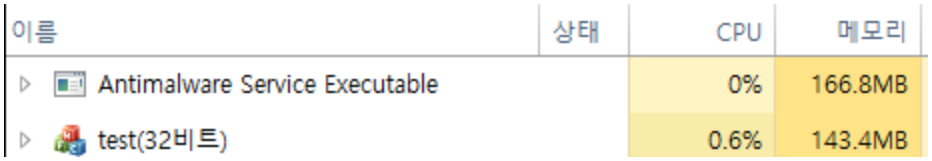
Ex) uGL-32-Complete-MultiObject-B

```
void uWnd::Draw()
{
    glClearColor(info.r,info.g,info.b,info.a);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // draw objects
    //for (int i=0;i<objs.GetSize();i++)
    //objs[i]->Draw();

    uObj *p = objs[0];
    float R= 80;
    float dq = (360./2000);

    for (int i=0;i<2000;i++)
    {
        float r=0,g=0,b=0;
        g = i;
        g = g/2000.;
        p->diffuse  = uColor(2*r,2*g,2*b);

        float x = R*cos(RAD(dq*i));
        float y = R*sin(RAD(dq*i));
        p->Trans(x,y,0);
        p->Draw();
    }

    glFinish();
}
```

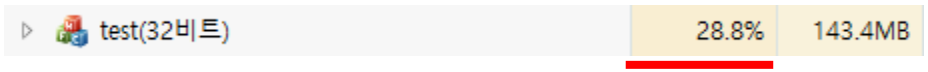
# Memory Usages

Ex) uGL-32-Complete-MultiObject-A

Idling mode

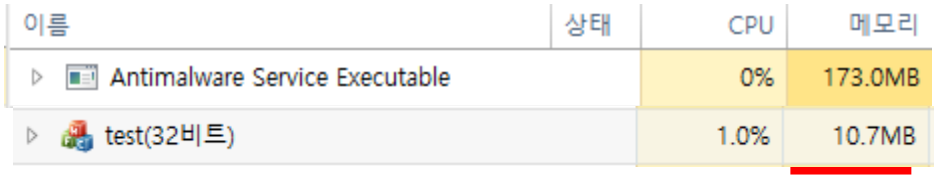



Rotation by Mouse move

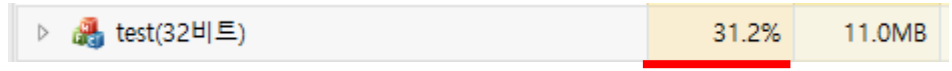



Ex) uGL-32-Complete-MultiObject-B

Idling mode




Rotation by Mouse move




| | CPU (Idle) | CPU (Work) | Memory |
|---|---|---|---|
| Ex) 32~A | 0.6% | 28.8% | 143.4Mb |
| EX) 32~B | 1.0% | 31.2% | 10.7Mb |

- Case A uses more memory.
- Case B uses more CPU, which is slightly consumption.