Sessió 1 - Effect Plugins

Introducció

En aquests exercicis us demanen que implementeu plugins (en C++) pel visualitzador bàsic de l'assignatura. Alguns exercicis requereixen escriure shaders (en GLSL).

Farem servir aquesta nomenclatura:

- **Effect plugin:** plugin que implementa els mètodes **preFrame** i/o **postFrame** (a banda d'altres mètodes que pugueu necessitar). Teniu un exemple a **plugins/effectcrt.**

- **Draw plugin**: plugin que implementa el mètode **drawScene** (a banda d'altres mètodes que pugueu necessitar). Teniu un exemple a **plugins/drawvbong.**

- **Action plugin:** plugin que implementa mètodes d'entrada com ara **keyPressEvent** (a banda d'altres mètodes que pugueu necessitar). Teniu un exemple a **plugins/navigatedefault**

- **Render plugin:** plugin que implementa el mètode **paintGL**, (a banda d'altres mètodes que pugueu necessitar). Teniu un exemple a **plugins/renderdefault.** Per desenvolupar shaders a la primera meitat del curs heu fet servir el plugin que podeu trobar a **plugin/shaderLoader/**

- VS, GS i FS fan referència a vertex shader, geometry shader i fragment shader, resp.

Warming up

Desplega les fonts del visualitzador (viewer) en un directori teu i prova construir els binaris, seguint les instruccions que us hem donat. Executa el viewer i prova de carregar algun model (en format .obj).

ModelInfo (1)

Escriu un **Effect Plugin** que escrigui (**postFrame**) la següent informació sobre l'escena: número total d'objectes carregats, número total de polígons, número total de vèrtexs, i el percentatge de polígons que són triangles. Feu una implementació que tingui un impacte negligible en el frame rate. Teniu un exemple de recorregut de l'escena a **plugins/drawvbong**

En aquesta versió només cal mostrar aquests valors pel canal de sortida estàndard (cout...).

Animate vertices plugin

Escriu un **effect plugin** que activi un VS per tal d'obtenir el mateix efecte de l'exercici **Animate Vertices (1).** Podeu mirar **plugins/effectcrt** com a exemple.

El mètode **onPluginLoad** haurà de carregar, compilar i muntar el shader. El mètode **preFrame**() els haurà d'activar i donar un valor apropiat a **l'uniform float time** (podeu feu servir timers de Qt) i a la resta d'uniforms que feu servir; el mètode **postFrame**() els haurà de desactivar.

ModelInfo (2)

Escriu un **Effect Plugin** que escrigui la mateixa informació que ModelInfo (1), però en aquesta versió cal mostrar aquestes dades per pantalla. Podeu mirar **plugins/showhelpNg** per veure l'ús de QPainter i QFont per dibuixar text en la finestra OpenGL.

Framerate

Escriu un **Effect Plugin** que mostri el *Frame Rate* actual de l'aplicació. Pots fer servir un `QElapsedTimer` per a calcular el temps usat en dibuixar els *frames*. Una alternativa és incrementar un comptador de frames a preFrame o postFrame, i crear un QTimer que emeti un signal cada segon, connectat a un slot que copiï el valor d'aquest comptador a la variable que conté els fps.

Podeu mostrar el resultat (per exemple, cada segon) pel cout.

Il·luminació per fragment amb shaders

Escriu un **effect plugin** que activi un VS i un FS per tal de tenir il·luminació de Phong per fragment. El mètode onPluginLoad haurà de carregar, compilar i muntar els shaders. El mètode preFrame() els haurà d'activar, i el mètode postFrame() els haurà de desactivar.

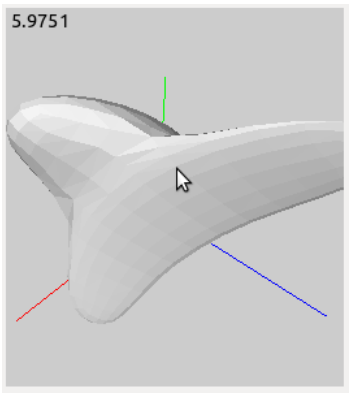
Recordeu enviar tots els uniforms que usin els shaders.

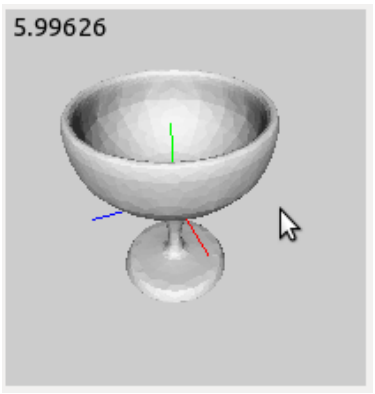
Show-degree

Escriu un **effect plugin** que escrigui el grau mig dels vèrtexs de l'escena (podeu assumir que l'escena contindrà **un únic objecte**). El grau d'un vèrtex és, simplement, el número de cares que incideixen en el vèrtex (el número de cares que fan servir el vèrtex). Per exemple, en un cub de sis quads, els vèrtexs tenen grau 3.

El mètode **onPluginLoad**() haurà de calcular el grau mig del primer objecte de l'escena. El mètode **postFrame**() l'haurà de mostrar a la finestra OpenGL.

Aquí teniu un exemple del resultat esperat amb els models boid.obj i glass.obj.





Gràfics 2017/18 Sessió 2 - Draw Plugins

Exercicis VBO

1. Draw bounding box

Escriu un plugin que pinti la capsa englobant de tots els objectes de l'escena.

El mètode **onPluginLoad** haurà de generar un VAO i els buffers necessaris per definir les sis cares (dotze triangles) d'una capsa "normalitzada", amb vèrtexs extrems (0,0,0) i (1,1,1). També haurà de carregar un parella VS+FS senzilla per pintar la capsa. Feu servir una variable **uniform** per tal de que el vostre VS pugui escalar i traslladar la capsa com convingui.

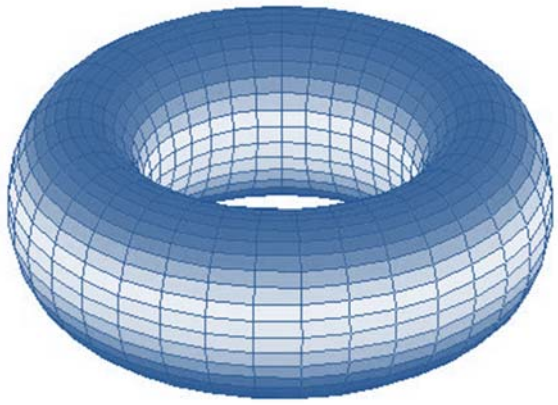
El mètode **postFrame** haurà de recórrer tots els objectes de l'escena i haurà de pintar la capsa de cada objecte fent servir el VAO i els shaders que haureu definit abans.

2. Draw smooth

En general, per una malla triangular de V vèrtexs (per exemple, un cub de V=8), es genera un VBO de V' vèrtexs, on V' és més gran o igual que V.

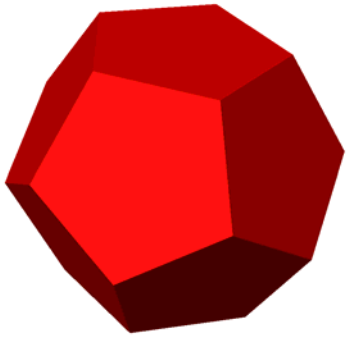
Podem considerar múltiples opcions:

- **V'=V.** Això implica que cada vèrtex (x,y,z) apareix un sol cop al VBO (sense repetits), amb una única normal. Per tant, totes les cares que fan servir aquest vèrtex el rebran amb la mateixa normal. El mateix s'aplica a la resta d'atributs (color, coordenades de textura...). Per alguns objectes suaus (per exemple, un torus) això pot no representar cap inconvenient.

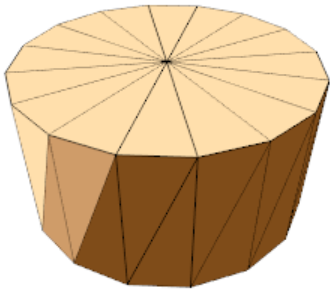


Però si l'objecte té arestes ben definides (com per exemple un cub) o en general presenta discontinuïtats als atributs, aquesta no és una bona opció.

- **V'=3T**. En aquest cas, per cada triangle, donem els seus tres vèrtexs. Això fa que un mateix vèrtex (x,y,z) aparegui múltiples cops al VBO: hi serà tantes vegades com triangles hi incideixen. En una malla de triangles tancada, el grau mig dels vèrtexs és aproximadament 6. Per tant, aquesta opció requereix molt més espai als buffers, però té l'avantatge que un mateix vèrtex pot tenir diferents atributs segons la cara que el fa servir (direm que tenim atributs **per-corner**). En el cas d'un cub, volem que cada vèrtex tingui la normal que li correspon segons la cara. Per tant, aquesta és la millor opció pels objectes amb totes les arestes ben definides.



- **V≤V'≤3T**. La majoria de malles de triangles tenen una combinació d'arestes suaus i arestes ben definides. En aquest cas, un vèrtex de grau *n* (compartir per *n* cares) pot aparèixer al VBO repetit 1, 2, ... *n* cops, depenent de com siguin les normals (i altres atributs) que necessiten rebre les *n* cares que usen el vèrtex. Per exemple, els vèrtexs del cilindre de la figura necessiten una normal vertical per les cares de les parts superior i inferior, i una única normal promig per les cares laterals. Aquesta opció repeteix al VBO només els vèrtexs que ho necessiten (perquè tenen atributs diferents) i per tant és la opció més general.



El que us demanem en aquest exercici és que, prenent com a punt de partida **drawvbong**, creeu un plugin que implementi la primera opció, és a dir, els vèrtexs no estaran duplicats al VBO; cada vèrtex apareixerà amb una única normal (mitjana de les normals que hi incideixen). Això implica usar **glDrawElements** (no **glDrawArrays**)

Sessió 3 - Render Plugins

✓ Multitexturing

Un dels plugins que et proporcionem és **multitex.** El que fa aquest plugin és, bàsicament:

- Al mètode **onPluginLoad**, crea un VS i un FS senzills per usar textures. El VS calcula de forma automàtica coordenades de textura (s,t) a partir de les coordenades dels vèrtexs i del radi d'una esfera contenidora del model (el radi el rep amb un uniform). El FS simplement assigna al color del fragment el color mig de dues textures (sampler0 i sampler1). El mètode també té el codi necessari per carregar dues imatges (obrirà un diàleg per que trieu les imatges) i definir un parell de textures. La classe QImage de Qt simplifica molt aquesta tasca. Observa que el codi està carregant la primera textura a la texture unit 0, i la segona a la texture unit 1 (crides **glActiveTexture**). Els identificadors OpenGL de les textures (textureId0,textureId1) són atributs de la classe, ja que més endavant seran necessaris per activar aquestes textures. La funció més rellevant és **glTexImage2D**, la qual defineix una textura a partir d'un buffer que conté les dades de la imatge que hem carregat:

```
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, image.width(), image.height(), 0,
              GL_RGBA, GL_UNSIGNED_BYTE, image.bits());
```

- Al mètode **preFrame**, s'activen el VS i el FS, i es defineixen els uniforms que contenen els texture units de cada sampler:

```
program->setUniformValue("sampler0", 0); // texture unit del primer sampler
program->setUniformValue("sampler1", 1);  // texture unit del segon  sampler
```

També s'activen les textures corresponents:

```
glActiveTexture(GL_TEXTURE0);  // texture unit 0
glBindTexture(GL_TEXTURE_2D, textureId0);
glActiveTexture(GL_TEXTURE1); // texture unit 1
glBindTexture(GL_TEXTURE_2D, textureId1);
```

En aquest exercici només et demanem que provis aquest plugin amb el viewer. Hauries d'obtenir imatges similars a aquestes (depenent de les textures que carreguis):

Gràfics
Curs 2019/20

Texture Splatting (plugin+shaders)

Escriu un **plugin** (a partir del multitex) que generi un efecte similar al que us demanàvem a l'exercici **Texture Splatting**.

A diferència del multitex, ara caldrà carregar **tres textures diferents**.

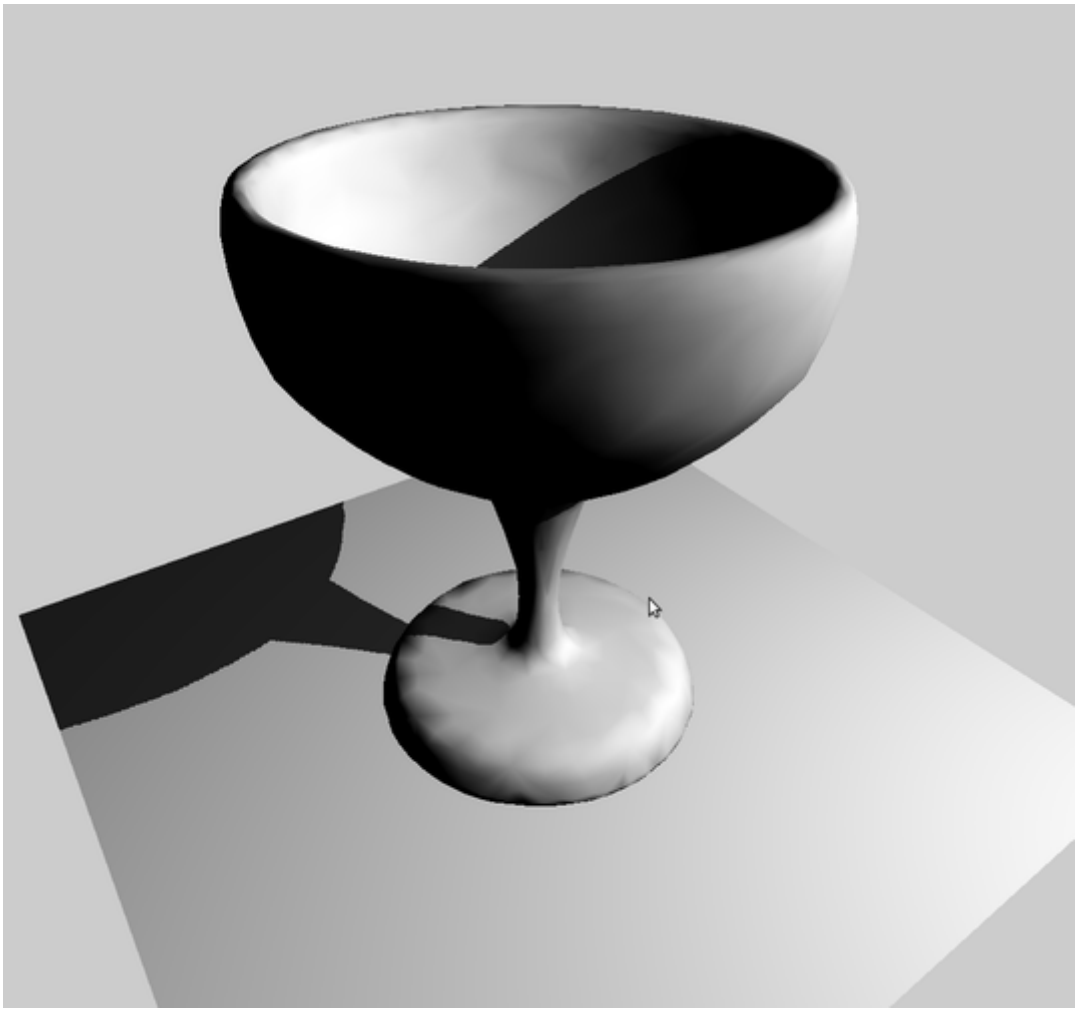
Aquí teniu un exemple (només orientatiu):

Gràfics
Curs 2019/20

ShadowMapping

Un dels plugins que et proporcionem és **shadowmap**.

En aquest exercici només et demanem que el provis amb el viewer. Hauries d'obtenir imatges similars a aquestes:



✓ Glowing

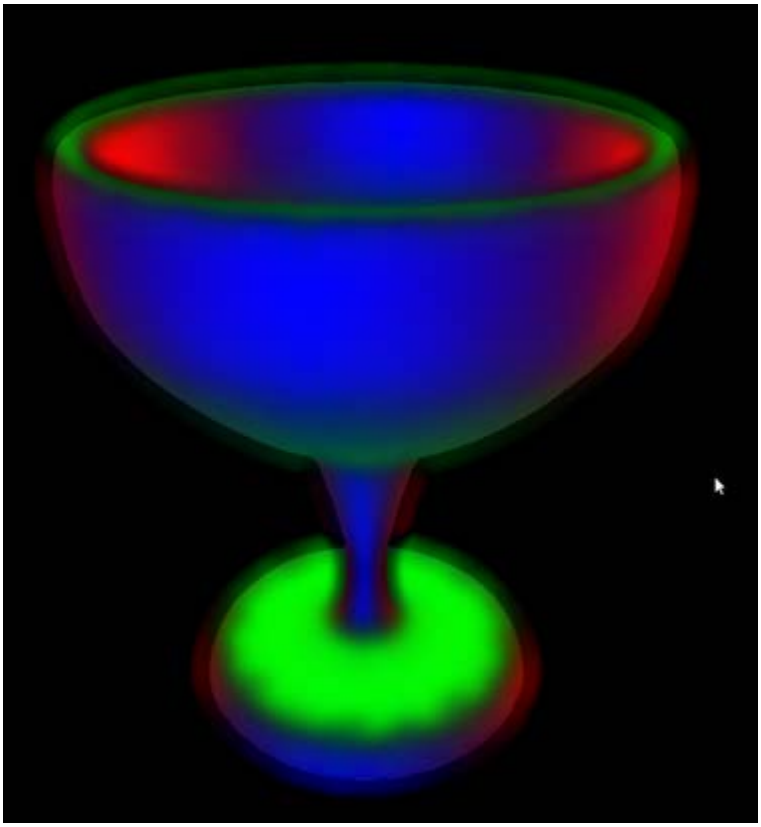
Un altre **plugin** que et proporcionem és **glowing**. Aquest plugin és un exemple de tècnica que requereix **múltiples passos de rendering**. En aquest cas, el mètode **paintGL** que implementa el plugin fa bàsicament aquests passos:

1. Pinta l'escena normalment (glClear, drawScene...), i **guarda el contingut del buffer de color en una textura** que hem inicialitzat prèviament. D'aquesta manera la textura té una còpia de la imatge corresponent al frame actual.

2. Activa un VS i un FS que implementen un efecte de glowing (el color de cada punt es veu afectat per la brillantor dels veïns), i dibuixa un rectangle que omple tot el viewport (amb vèrtexs en clip space de -1 a 1). Per a cada fragment, el FS consultarà diferents texels de la textura per produir aquest efecte.

En general, per dibuixar quelcom en una textura és més eficient utilitzar Frame Buffer Objects (FBOs). En aquest exemple però, per tal de fer-ho més senzill, hem dibuixat igualment al *on-screen* frame buffer, i després hem utilitzat **glCopyTexSubImage2D** per copiar el contingut del buffer de color a una textura que hem inicialitzat prèviament. Amb el mateix objectiu de fer l'exemple senzill, veureu que com es carrega el plugin, la finestra OpenGL es canvia a una mida potència de dos (512x512, 1024x1024...).

En aquest exercici només et demanem que el provis amb el viewer. Hauries d'obtenir imatges com aquesta:



Distort

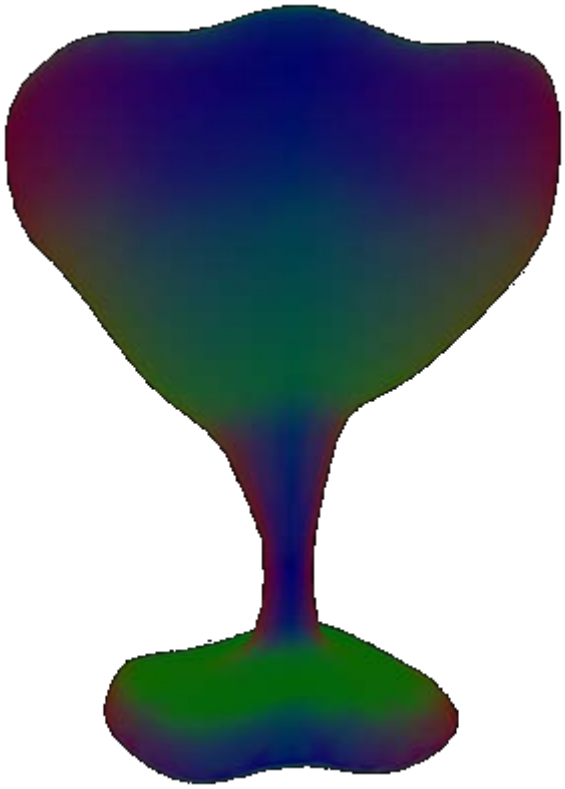
Prenent com a punt de partida el plugin de glowing, escriu un plugin que, amb un VS i un FS apropiats, aconsegueixi un efecte d'ones aplicat a la imatge.

Per aconseguir aquest efecte, caldrà dos passos de rendering.

Al primer pas, es dibuixarà l'escena normalment per obtenir una textura.

Al segon pas, es dibuixarà un rectangle amb un FS que calcularà el color final accedint a la textura amb coordenades de textura (s,t) alterades afegint-hi un offset 0.01*sin(10.0*time + 30.0*s).

Aquí teniu un exemple aproximat del resultat esperat:



Gràfics
Curs 2019/20

Reflection

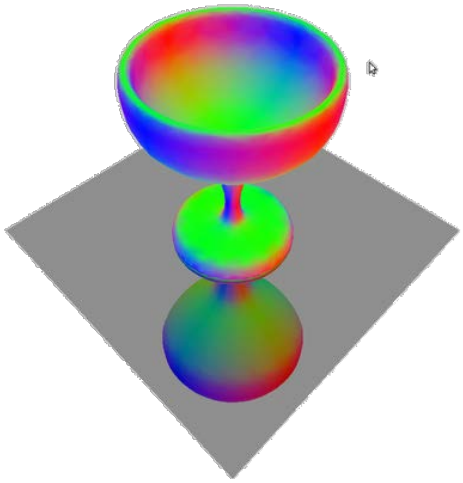
Escriu un plugin (prenent com a base el glowing) que implementi la tècnica de reflexió amb textures dinàmiques. Com a mirall, podeu fer servir una de les cares de la capsa contenidora de l'escena. Caldrà fer tres tasques al paintGL:

1. Dibuixar l'escena reflectida respecte el pla del mirall. Guardar la imatge resultant en una textura i esborrar el framebuffer.

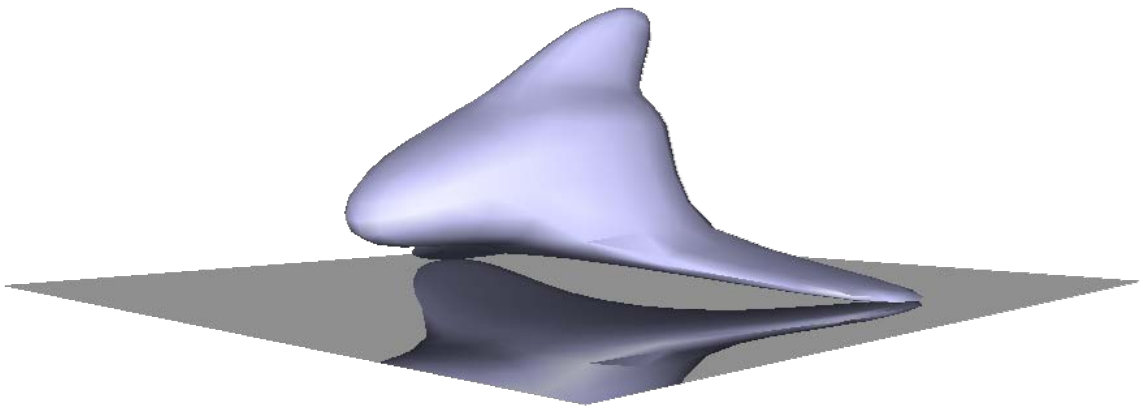
2. Esborrar el framebuffer i dibuixar l'escena en la posició real.

3. Dibuixar el quad del mirall, texturat amb la textura del primer pas.

Aquí teniu alguns exemples:



Observa que l'objecte reflectit només apareix a la porció del pla ocupada pel mirall:



Gràfics
Curs 2018/19

Sessió 4 - Action Plugins

Resaltat de l'objecte seleccionat

Escriu un **effect plugin** que resalti d'alguna manera l'objecte seleccionat, si n'hi ha cap (per exemple, pintant la seva capsa englobant en filferros). Caldrà que ho feu re-implementant el mètode **postFrame**().

L'index de l'objecte seleccionat es pot obtenir amb el mètode **selectedObject**() de la classe **Scene** (vigileu que un valor -1 indica que no hi ha cap objecte seleccionat):

```
int seleccionat = scene()->selectedObject();
```

Per defecte, l'objecte seleccionat és el primer (índex 0), si s'ha carregat algun objecte.

Selecció d'objectes per teclat

Escriu un **action plugin** per tal que quan l'usuari premi una tecla 0..9, es seleccioni l'objecte corresponent de l'escena (si existeix). Per exemple, amb '0' es seleccionarà el primer objecte del vector d'objectes de l'escena.

Per tal d'aconseguir això, haureu d'implementar el mèdode **keyPressEvent**(). Aquest mètode haurà d'establir l'objecte seleccionat amb el mètode **setSelectedObject** de la classe **Scene**, si aquest existeix, i després cridarà a **update** per tal que es repinti l'escena.

Si l'usuari prem una tecla 0..9 que no correspon a cap objecte (per exemple, '5' quan l'escena només té 3 objectes), cal posar l'objecte seleccionat a -1.

Per tal de provar el funcionament correcte d'aquest plugin, cal combinar-lo amb el plugin de l'exercici anterior que indiqui d'alguna manera l'objecte seleccionat.

Selecció d'objectes amb el mouse

Escriu un **action plugin** per tal que quan l'usuari faci clic amb el mouse (per exemple, LMB + Ctrl), es seleccioni l'objecte visible més proper a l'observador que estigui sota el cursor (si n'hi ha cap, és clar).

El mètode **onPluginLoad** carregarà un VS i FS minimalistes. El VS simplement escriurà gl_Position com és habitual. FS simplement escriurà com a color el valor d'una variable uniform que rebrà de l'aplicació,

```
uniform vec4 color;
```

El mèdode **mouseReleaseEvent**() implantarà la selecció pròpiament dita. Aquest mètode haurà de:

(a) Comprovar que efectivament s'ha fet click amb el botó adient i els modificadors (Shift, Control...) adients. Per exemple:

```
if (! (e->button() & Qt::RightButton)) return;
if ( e->modifiers() & (Qt::ShiftModifier)) return;
if (! (e->modifiers() & Qt::ControlModifier)) return;
```

(b) Esborrar els buffers amb un color de fons únic (ex. blanc).
(c) Activar (bind) el shader program amb el VS+FS d'abans.
(d) Enviar els uniforms que facin servir els vostres shaders.
(e) Pintar l'escena assegurant-se que cada objecte es pinta amb un color únic que permeti identificar l'objecte (i diferent del color de fons); això ho fareu definint el uniform vec4 color que farà servir el FS:

```
// per cada objecte
for (unsigned int i=0; i<scene()->objects().size(); ++i)
{
   GLubyte color[4];
   encodeID(i,color); // trieu la conversió que volgueu
   program->setUniformValue("color", QVector4D(color[0]/255.0,
color[1]/255., color[2]/255., 1.0));
   drawPlugin()->drawObject(i);
}
```

(f) Llegir el color del buffer de color sota la posició del cursor,

```
int x = e->x();
int y = glwidget()->height()-e->y();
GLubyte read[4];
glReadPixels(x, y, 1, 1, GL_RGBA, GL_UNSIGNED_BYTE, read);
```

(g) Obtenir l'identificador de l'objecte corresponent i, si no és color de fons, establir l'objecte seleccionat amb el mètode **setSelectedObject** de la classe **Scene**,
(h) Cridar a **update** per tal que es repinti l'escena.