


Université Gustave Eiffel

École Supérieure d'Ingénieurs Paris-Est

5 boulevard Descartes · 77420 Champs-sur-Marne

Réalisé dans le cadre de la

Formation IMAC  — 2^{ème} année du cycle ingénieur

Pour les cours de **Synthèse d'image**, **Architecture logicielle** et **C++**

Professeurs référents : M. Venceslas Biri, M. Steeve Vincent, M. Sylvain Cherrier et M. Vincent Nozick

RAPPORT DE PROJET

Conception et programmation d'un jeu vidéo contemplatif en C++ avec OpenGL

Alexandre Escudero · aescuder@etud.u-pem.fr

Ludwig Chieng · lchieng@etud.u-pem.fr

Amélie Crouigneau · acrouign@etud.u-pem.fr

03/01/2021



Table des matières

Introduction	3
1. Organisation	3
2. Développement	3
2.1. Arborescence des fichiers	3
2.2. Architecture logicielle	3
2.2.1. Séquence d'initialisation	3
2.2.2. Séquence de boucle	3
2.2.3. Description rapide des classes	4
2.2.4. Diagramme de classe	4
2.2.5. Design patterns utilisés	4
2.3. C++	4
2.4. Synthèse d'image	4
2.4.1. Trackball camera	4
2.4.2. Modèle de mécanique du solide	5
2.4.3. Modèle d'illumination	5
2.4.4. Skybox	5
2.4.5. Texturing procedural	5
3. Démo	6
4. Améliorations et conclusions	7
4.1. Architecture	7
4.2. C++	7
4.2. Synthèse d'image	7
Annexes	8
Exemple d'un fichier config	8
Extrait d'un fichier descripteur de scène	8
Shaders map de la Pentaball	9
Diagramme de classes	10

Introduction

Notre projet a abouti au développement d'un jeu vidéo appelé *Lihowar*, réalisé avec des technologies C++, OpenGL et SDL2. Le gameplay se veut contemplatif. Le joueur contrôle un dirigeable dans lequel il se déplace d'îles en îles. Chaque île majeure possède un *beacon* qu'il faut allumer en activant des éléments simples (plateformes...). Dans la version du jeu rendu avec ce rapport, la fonctionnalité est partiellement implémentée. [\[documentation en ligne\]](#) [\[démonstration sur youtube\]](#)

L'équipe de développement se compose d'Alexandre Escudero, Amélie Crouigneau et Ludwig Chieng.

1. Organisation

Il a été difficile pour nous de trouver une méthode de travail, car aucun de nous n'avait déjà conçu une application OpenGL d'une telle ampleur. La conception logicielle s'est donc faite par tâtonnement. En effet, inutile de commencer à esquisser un diagramme de classe complet quand on sait à peine manipuler un VBO.

Nous nous sommes inspirés de la méthode agile, c'est-à-dire du développement incrémental et avoir une application fonctionnelle plutôt qu'exhaustive.

Au départ, nous n'avons pas vraiment de repère sur comment architecturer l'application. Avec l'expérience du développement précédent d'Imac Wars, on sait que le MVC strict ne fonctionne pas. On adopte l'architecture classique avec SDL où chaque objet du jeu possède ses données et sa méthode `render()`.

2. Développement

2.1. Arborescence des fichiers

[\[Repo git en ligne\]](#)

assets	textures, sons et modèle 3D
config	fichiers de config JSON
doc	répertoire Doxygen
glimac	lib glimac
lihowarlib	moteur de jeu
misc	fichiers divers
screenshots...	
scenes	fichiers JSON de scenes de jeu
src	
shaders	GLSL shaders
main.cpp	
third-party	
include	
glm	
tao	lib hpp only pour JSON
tools	mini-programmes utilitaires
normalShifter.cpp	transforme une normal map

(exemples de fichiers JSON en annexe)

2.2. Architecture logicielle

2.2.1. Séquence d'initialisation

(s) : singleton

```
main
-> glimac::SDLWindowManager
-> Game (s)
    -> GameController (s)
        -> SceneSerializer::load() (s)
            -> AssetManager (s)
                -> collection of Mesh
                    -> foreach required Mesh
                        -> load .obj file
                -> collection of Texture
                    -> foreach required Texture
                        -> load texture
            -> Scene
                -> Skybox
                -> Player
        -> load objects and their subobjects into Scene
        -> load islands and their subobjects into Scene
    -> GameRenderer (s)
        -> TrackBallCamera
-> init lights
```

2.2.2. Séquence de boucle

```
main loop:
-> handle mouse/keyboard event
-> Game::update()
    -> Pump and process SDL events (keyboard and joystick)
    -> GameController::update()
        -> update skybox position
        -> update objects dynamics (apply forces or torques)
        -> update scene objects
-> Game::render()
    -> GameController::render()
        -> GameRenderer::render(scene)
        -> render each scene objects
            -> update model and view matrices
            -> bind variables to GPU as uniform
            -> draw object
                -> draw itself
                -> draw its subobjects
-> swap buffers
```

2.2.3. Description rapide des classes

Le détail est disponible dans la [documentation](#) Doxygen.

Game : classe mère

GameController : gère le traitement de ce qui se passe in-game

GameRenderer : gère l'écriture sur le framebuffer, communication avec le GLSL program...

AssetManager : initialise et charge en mémoire les textures et les modèles 3D

Mesh : correspond à un fichier OBJ

Object : objet qui apparaît dans le jeu

ObjectDynamic : implémente la mécanique de Newton

Object possède une liste d'*Object* appelé *subobjects*. L'application gère les objets de manière récursive.

2.2.4. Diagramme de classe

(voir en Annexe)

2.2.5. Design patterns utilisés

Singleton. Game, GameController, GameRenderer...

Observer. Une *TrackballCamera* observe un *Object* : lorsque l'*Object* change, la trackball camera se met à jour

Iterator. Avec les list et vector de la STL

Forte cohésion. ex. : Light, LightDirectional, LightPoint ; ou encore Object, ObjectDynamic, Player

Chaîne de responsabilité. la méthode render() est appelé initialement par le main

- l'instance *Game* délègue à *GameController*
- l'instance *GameController* transmet la *Scene* et délègue à *GameRenderer*
- l'instance *GameRenderer* active le programme GLSL adéquat et délègue à chaque *Object* la responsabilité de se dessiner
- l'instance *Object* se dessine elle-même et délègue récursivement chacun de ses sous-*Object* la responsabilité de se dessiner

2.3. C++

Fonctions template

- `Object::findAll<T>()` permet de trouver récursivement tous les *Objects* de type T (sous-classe d'*Object*) parmi les sous-objets d'un *Object*
- `Serializer::get<T>` permet de récupérer une donnée dans un JSON, casté en type T

Polymorphisme

- la classe Object (sous-classes : Beacon, Platform, Pentaball...)
- la class Light (sous-classes : Light Directional, Light Point)

Outil de la STL. vector, list, map, string, exception

Exception. *LihowarException* pour discriminer les erreurs provenant de Lihowar des erreurs standards ; *LihowarIOException* pour les erreurs impliquant la sérialisation de fichier (*ConfigSerializer* ou *SceneSerializer*)

Exemples de gestion des exceptions :

- `Texture::getPath(TextureName tn)` permet de récupérer le chemin d'accès à une image à partir d'un élément de l'enum TextureName. La méthode génère une exception si la TextureName n'a aucun chemin d'accès associé
- `GameConfig::load()` charge un fichier de configuration JSON défini par l'utilisateur en tant qu'argument de l'exécutable. Si ce fichier n'existe pas, une exception est générée puis traitée : on bascule sur le fichier de configuration default.json. En dernier recours, si celui-ci est aussi introuvable, alors, on passe sur une configuration hardcoded dans GameConfig.cpp

Assert. ex. : `AssertManager::checkAssets()` vérifie que les éléments dans les enums `TextureName` et `MeshName` ont bien tous une entrée dans `Texture::PATHS` et `Mesh::PATHS`. Dans le cas contraire, le développeur a fait une erreur.

Namespace. On a l'espace *lihowar* et l'espace *dp* pour les classes abstraites du design pattern Observer

Fonctions lambda. ex. : Dans *TrackballCamera* : lerp, easeQuad et easeCos, sont des fonctions maths qui permettent d'adoucir les mouvements de caméra.

Fonctions variadics. ex. : `glimac::SDLWindowManager::isKeyPressed()` prend en paramètre une ou plusieurs touches de clavier et renvoie true si au moins l'une d'elle est activée.

2.4. Synthèse d'image

2.4.1. Trackball camera

[\[documentation en ligne\]](#)

La classe implémente des fonctionnalités relativement avancées.

Lorsque le joueur avance, la caméra recule légèrement pour donner une impression d'accélération (on a appelé ça le kickback). Cela est géré par l'attribut `_posOffset`.

Également, lorsqu'il tourne, la position de la caméra varie légèrement (l'angle autour de Y) pour éviter l'effet "caméra collée au capot" et rendre le suivi cam plus doux. Cela est géré par l'attribut `_angOffset`.

La position de la caméra sur son axe Z (distance entre la caméra et sa cible), est contrôlée par la molette de la souris. Pour éviter de buter trop violemment contre la "distance max" ou la "distance min", la distance ne varie pas linéairement. C'est ici qu'on utilise les fonctions maths de *easing*.

2.4.2. Modèle de mécanique du solide

Implémentée par la classe *ObjectDynamic* :
[documentation en ligne]

Un *ObjectDynamic* a une accélération (`_acc`), une vitesse (`_vel`) et une position (`_prs.pos`), qui matérialise son déplacement au cours du temps.

Il possède également une accélération angulaire (`_angAcc`), une vitesse angulaire (`_angVel`) et une position angulaire (`_prs.rot`).

Le principe fondamental de la dynamique de Newton :

$$F = ma \Leftrightarrow F/m = a$$

```
_acc += force / _mass;
```

Par extrapolation et par simplification, on fait de même pour l'accélération angulaire :

```
_angAcc += torque / _inertia;
```

L'inertie est calculée en fonction de la masse et des dimensions d'un objet selon X, Y et Z. C'est pour cela que l'inertie est un `vec3`. Dans la modélisation de Newton, l'inertie est représentée par une matrice d'inertie qui dépend de la forme de l'objet. Ici, on se contente de la bounding box.

ObjectDynamic implémente aussi deux méthodes pour appliquer une force de traînée "linéaire" et "angulaire" à chaque tour de boucle pour éviter que les objets en mouvement ne le restent indéfiniment, comme dans l'espace.

2.4.3. Modèle d'illumination

Le shader *multilights.fs.glsl* gère deux tableaux de `LightDir` et de `LightPoint`.

On reprend le modèle de Blinn-Phong, avec une diffuse map, une specular map, une luminance map, une ambient occlusion map et une normal map.

On a aussi une lumière ambiante pour déboucher les tons foncés.

Le blend mode de la luminance map :

```
fFragColor = max(fFragColor, uKl * uLuminMap);
```

Workflow pour faire les shader maps :

- Réalisation du mesh sous Cinema4D ou Blender
- Réalisation des diffuse, specular, luminance, normal maps (procédural ou image stock)
- UV mapping
- Calcul de l'occlusion ambiante

(Exemple de shaders map en annexe)

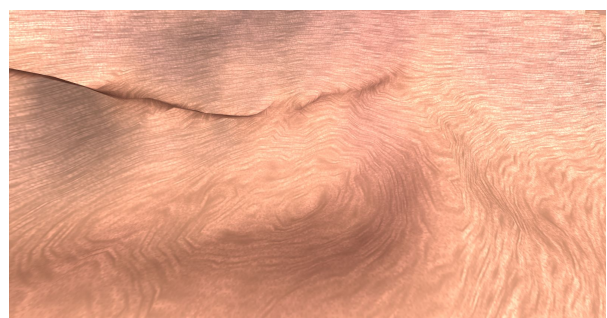
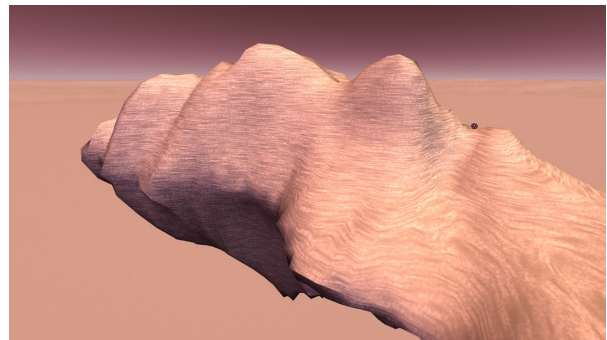
2.4.4. Skybox

On n'a su que bien après l'avoir implémenté qu'il existait les textures *CubeMap* sous OpenGL, mais nous sommes quand même restés sur une texture classique avec un mesh cube fait à la main.

La skybox a ses shaders attitrés. Le fragment shader renvoie directement la couleur du texel sans traitement. Le vertex shader décale très légèrement les faces du cube vers l'intérieur pour éviter d'avoir un gap entre les faces du cube, laissant apparaître une ligne noire.

2.4.5. Texturing procedural

En faisant des objets *Island* immense, on ne pouvait pas les texturer avec des images. On a donc décidé de le faire procéduralement sur le GPU. [source]



Étapes :

- Faire très légèrement varier la couleur en fonction de la normale du vertex
- Donner une couleur plutôt jaune/orange claire aux fragments orientés vers le haut (qui ont une normale en Y élevée)
 - ou bien -
- Donner une couleur plutôt gris foncé aux fragments orientés vers le bas (qui ont une normale en Y basse)

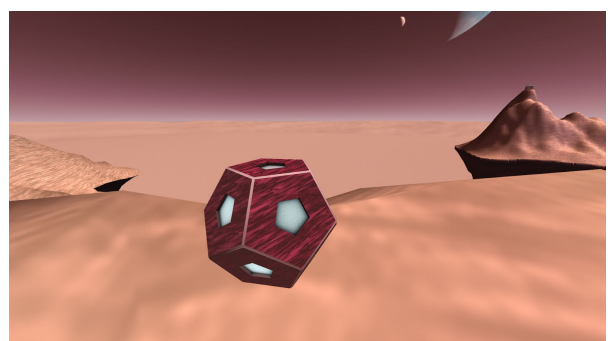
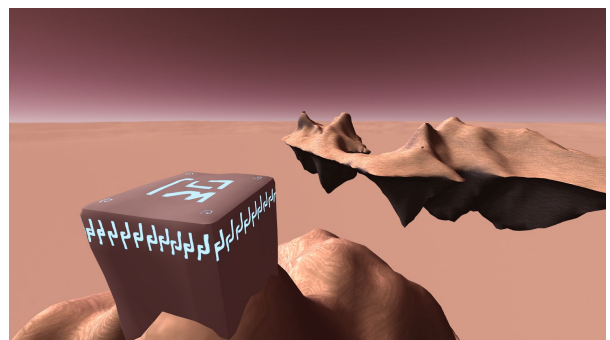
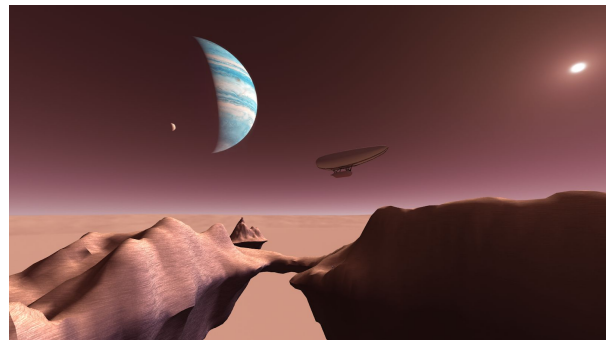
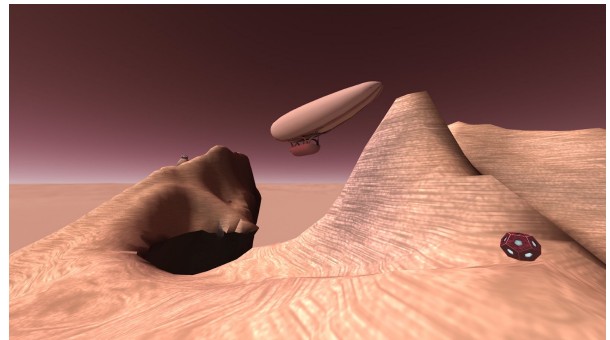
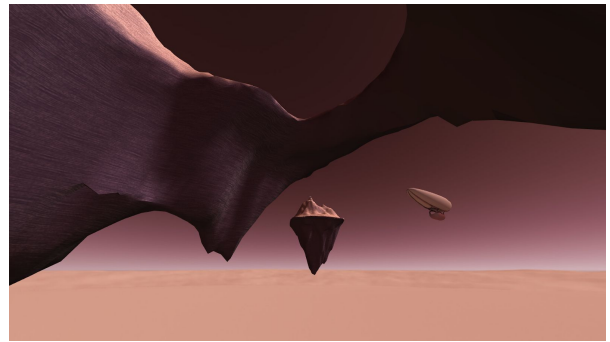
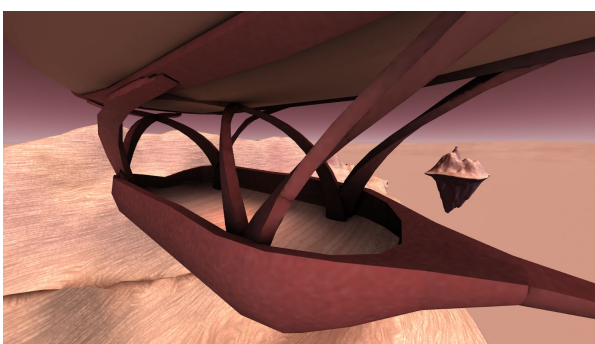
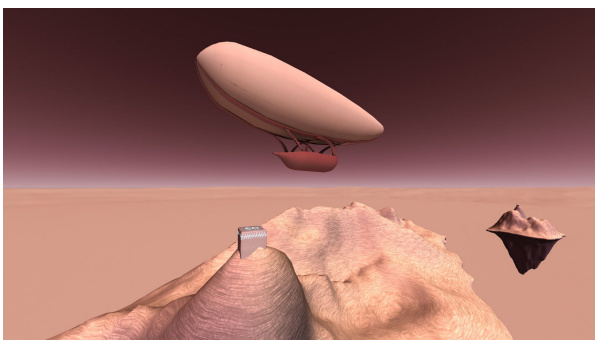
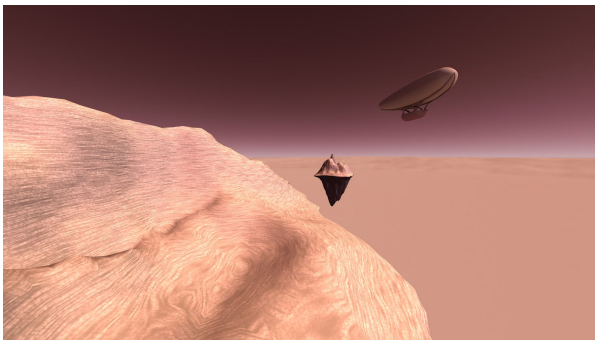
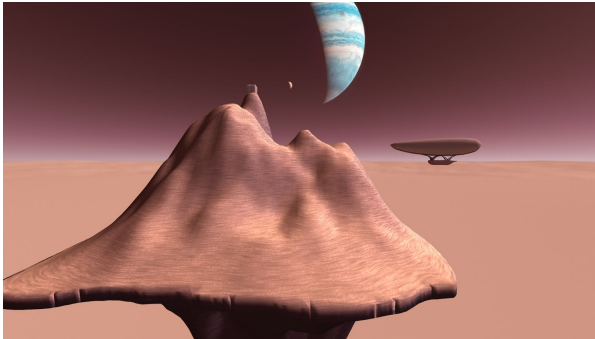
- Additive blending avec une couche nommée "sand". Le *sand* est constitué de deux couches de *noise* dont un est "aplati" sur l'axe Y, pour donner un effet de strate sédimentaire. Le *sand* agit comme une specular map :

```
sand = dot(fragNormal, -wo) ...  
...  
sand *= dot(halfV, fragNormal)
```

- Application des lumières Blinn-Phong

3. Démo

[\[démo sur youtube\]](#)



4. Améliorations et conclusions

4.1. Architecture

A ce stade du développement, on se retrouve bloqués, c'est pour cela qu'il n'y a pas d'interaction avec la scène. C'est bête parce que presque tout y est, les classes *Platform*, *Beacon* et *Pentaball* sont vides, elles n'attendent que d'avoir des méthodes spécialisées.

En fait, ce qui bloque c'est la chose suivante : on a mal géré l'architecture objet/sous-objets, on s'emmêle les pinceaux entre les repères des modèles, des objets, des sous-objets. Pour avoir la position d'un objet dans le repère global, il faut récupérer toutes les Model matrices de ses parents sauf qu'on n'avait pas prévu de navigabilité depuis les fils vers les parents. Bref, la galère. Dans l'idéal il aurait fallu repartir de zéro et commencer par coder un *TreeNode* d'*Object* navigable dans les deux sens, avec un calcul des Model matrices optimal et avec la possibilité de récupérer les coordonnées des sous-objets dans n'importe quel repère souhaité (repère joueur, repère global, repère du parent...).

On s'est bien débrouillé pour factoriser du code dupliqué, par contre on a eu du mal pour factoriser du code similaire, notamment la partie avec les GLSL program et les shaders map.

4.2. C++

Côté C++, on a toujours du mal avec les `unique_ptr` ou les `shared_ptr` et avec la gestion de la mémoire. Mais on est assez fier de la récursivité, des fonctions template, lambda et variadics.

4.2. Synthèse d'image

Il y a probablement de l'optimisation à faire côté fragment shader.

Globalement on est très satisfait de la partie synthèse d'image. On aurait aimé ajouter :

- du bump mapping pour le shader Island
- de la génération procédurale d'Island
- une évolution au cours du temps de la skybox (Soleil qui se couche et variation de la lumière du Soleil)
- du mipmap sur les textures
- du bloom et des lens flares
- des particules
- de la musique et des SFX (on a composé la musique mais on n'a pas réussi à faire fonctionner SDL_mixer 🤔🤔)

Annexes

Exemple d'un fichier config

```
{
  "debug": false,
  "path_assets": "assets/",
  "path_shaders": "shaders/",
  "path_scenes": "scenes/",
  "fullscreen": true,
  "window_width": 1920,
  "window_height": 1080,
  "max_framerate": 60,
  "min_fov": 70,
  "max_fov": 95,
  "z_near": 0.1,
  "z_far": 5000,
  "use_antialiasing": true,
  "msaa": 4,
  "scene": "scene1.json"
}
```

Extrait d'un fichier descripteur de scène

```
{
  "islands": [
    {
      "type": "Island",
      "meshName": "ISLAND1",
      "prs": {
        "pos": [70.0, -40.0, 0.0],
        "sca": [2.0, 2.0, 2.0]
      },
      "subobjects": [
        {
          "type": "Beacon",
          "meshName": "BEACON1",
          "material": {
            "diff": "BEACON1_DIFF",
            "lumin": "BEACON1_LUMIN"
          },
          "prs": {
            "pos": [-5.621, 28.893, 5.174],
            "rot": [180.0, 0.0, 0.0],
            "sca": [1.5, 1.5, 1.5]
          }
        }
      ]
    },
    {
      "type": "Island",
      "meshName": "ISLAND2",
      "prs": {
        "pos": [500.0, -40.0, 0.0],
        "sca": [4.0, 4.0, 4.0]
      }
    }
  ]
}
```

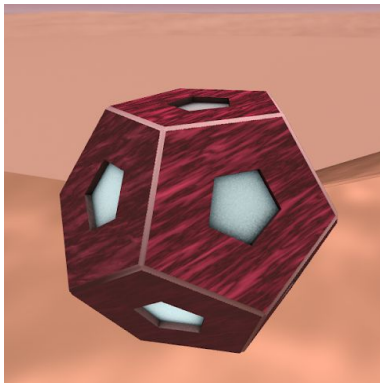


```

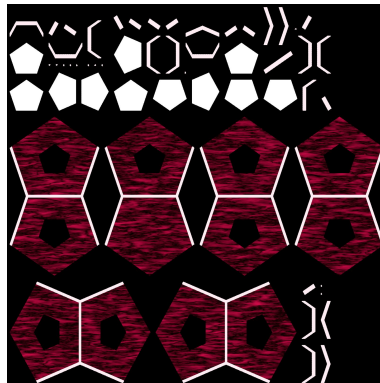
    },
    "subobjects": [
      {
        "type" : "Beacon",
        "meshName": "BEACON1",
        "material": {
          "diff": "BEACON2_DIFF",
          "lumin": "BEACON2_LUMIN"
        },
        "prs": {
          "pos": [51.521, 20.539, -35.154],
          "rot": [180.0, 0.0, 0.0],
          "sca": [1.5, 1.5, 1.5]
        }
      }
    ]
  },
  "objects": []
}

```

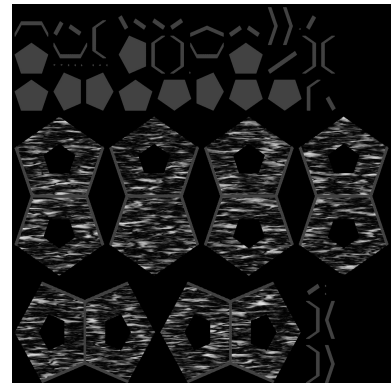
Shaders map de la Pentaball



Rendu



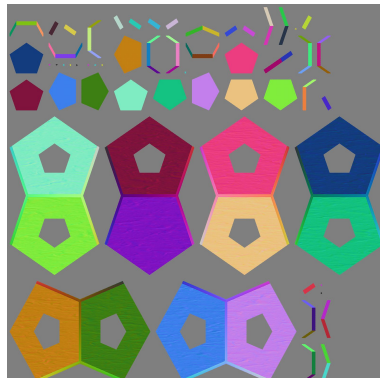
Diffuse map



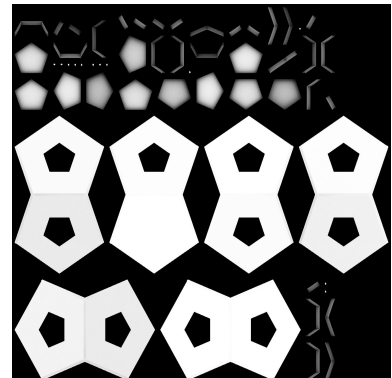
Specular map



Luminance map



Normal map



AO map

Diagramme de classes

