XVIII – EXTENSIONS DE BLENDER (Extending Blender)

A la différence de beaucoup de programmes qui peuvent vous être familiers, **Blender** n'est pas monolithique et statique. Vous pouvez étendre ses finctionnalités sans avoir à modifier le code source et à recompiler. Il existe deux façons d'étendre **Blender** : des scripts en **Python** ou des plugins binaires. La première méthode est celle qui est préférée et qui est le plus largement utilisée. Ce chapitre décrira les deux méthodes.

18.1. Les Scripts Intégrés (Bundled Scripts)

Comme vous devez déjà le savoir, **Blender** est fourni avec un interpréteur Python intégré, qui lui permet de lancer des scripts écrits dans ce langage. Ces scripts peuvent utiliser l'API **Blender Python** (**bpython**) pour accéder aux routines internes du programme et étendre ainsi les fonctionnalités de **Blender**.

Vous pouvez trouver des informations sur de nombreux scripts (et les télécharger) sur la page http://www.blender3d.org/cms/Python_Scripts.3.0.html . De plus, les créateurs de scripts annoncent et parlent de leurs créations sur les forums du site www.blender.org (**API development**) et en particulier sur le site www.elysiun.com (pour les utilisateurs).

Remarque : Une limitation existe car certains scripts intégrés ne peuvent pas fonctionner si vous n'avez pas installé une version complète du langage **Python** (actuellement en version **2.5**) avec tous les modules **Python**.

En cas de problème :
Même avec un code parfaitement testé, il est toujours possible que quelque chose ne fonctionne pas comme vous le souhaitez. Cela peut être une bogue, mais cela peut aussi arriver parce qu'il vous manque une information sur la façon d'utiliser le script ou parce que c'est toujours une version ancienne et incomplète avec des capacités limitées. Quel que soit le problème, la démarche à suivre suivante vous est recommandée :

- Lisez le Tooltip dans l'entrée de menu et, si elles sont disponibles, toutes les informations dans l'interface graphique (**GUI**) du script;
- Ouvrez le fichier du script lui-même (fichier **.py**) dans une éditeur de textes (celui de **Blender** par exemple); les auteurs y placent généralement des commentaires importants, ainsi que des options manipulables;
- Essayez une recherche en ligne sur les forums dédiés aux scripts des sites www.blender.org et/ou www.elysiun.com;
- Le fichier du script peut contenir l'email de l'auteur ou un lien vers un site pour le support;
- Si tout ceci échoue, postez votre question sur l'un des forums dont nous venons de parler.

Les Différentes Classes de Scripts pour Blender

Voici les différents "répertoires" dans lesquels vous pouvez retrouver les scripts fournis avec **Blender**, ou ceux que vous avez ajouté après les avoir téléchargés sur Internet.

Les sous-répertoires du menu Scripts de la fenêtre Scripts

Wizards
Ces Scripts sont disponibles dans la fenêtre **Scripts**, via l'option **Wizards** du menu **Scripts**. Ce menu, seulement présent dans la fenêtre **Scripts**, est l'emplacement pour les scripts **Python** les plus complexes, qui peuvent nécessiter plus de place pour leur interface (**GUI**) ou ne peuvent être définis dans une autre catégorie.

Images
Ces scripts peuvent être activés dans le menu **Image** de l'éditeur **UV/Image** (en mode **Image**).

UV
Ces scripts peuvent être activés dans l'option **UV** du menu **Scripts** de la fenêtre **Scripts**, ou dans le menu **UVs** de l'éditeur **UV/Image** (en mode **UV**).

Themes
Ces Scripts sont disponibles dans la fenêtre **Scripts**, via l'option **Themes** du menu **Scripts**.

Render
Ces Scripts sont disponibles dans la fenêtre **Scripts**, via l'option **Render** du menu **Scripts**.

Object
Ces scripts d'outils, destinés à automatiser certaines tâches répétitives, peuvent être activés avec l'option **Object** du menu **Scripts** de la fenêtre **Scripts**, ou par l'option **Scripts** du menu **Object** de la **Vue 3D** courante (en mode **Object**).

Misc
Ces Scripts sont disponibles dans la fenêtre **Scripts**, via l'option **Misc** du menu **Scripts**. Ici, sont placés les scripts particuliers, comme le script sur les fractales de Julia.

Mesh
Ces scripts de très bons outils de modélisation peuvent être activés avec l'option **Mesh** du menu **Scripts** de la fenêtre **Scripts**, ou par l'option **Scripts** du menu **Mesh** de la **Vue 3D** courante (en mode **Edit** (Mesh)).

Material
Ces Scripts sont disponibles dans la fenêtre **Scripts**, via l'option **Material** du menu **Scripts**.

Import
Ces Scripts sont disponibles dans la fenêtre principale, via l'option **Import** du menu **File** ou dans le menu **Scripts** de la fenêtre **Scripts**. Grâce à ses différents filtres d'import, **Blender** est capable de lire les formats de fichiers suivants, issus d'autres applications graphiques, 2D ou 3D.

Export
Ces Scripts sont disponibles dans la fenêtre principale, via l'option **Export** du menu **File** ou dans le menu **Scripts** de la fenêtre **Scripts**. Grâce à ses différents filtres d'export, **Blender** est capable d'écrire dans les formats de fichiers suivants, issus d'autres applications graphiques, 2D ou 3D.

Animation
Ces Scripts sont disponibles dans la fenêtre **Scripts**, via l'option **Animation** du menu **Scripts**.

Add
Ces Scripts sont disponibles dans la fenêtre **Scripts**, via l'option **Add** du menu **Scripts**.

Les sous-répertoires du Menu principal de Blender

Add Mesh
Ces Scripts sont disponibles dans le menu **Add > Mesh > …..** Des slots **Python** sont ici disponibles pour y ajouter des scripts personnalisés.

Add Armature
Ces Scripts sont disponibles dans le menu **Add > Armature**. Des slots **Python** sont ici disponibles pour y ajouter des scripts personnalisés.

Websites
Ces Scripts sont disponibles dans la fenêtre principale, via l'option **Websites** du menu **Help**. Ces scripts ouvrent le navigateur internet par défaut de l'utilisateur vers la destination choisie, en utilisant le module Python **webbrowser.py**. Suggestion : si il y a d'autres liens que vous voudriez y voir, dupliquez simplement l'un de ces scripts, modifiez son nom de fichier et, dans son contenu, modifiez le champ **Name** et l'adresse (**URL**).

Help
Ces Scripts sont disponibles dans le menu **Help** de la fenêtre principale.

System
Ces Scripts sont disponibles dans la fenêtre principale, via l'option **System** du menu **Help**.

Les sous-répertoires des Menus des différentes fenêtres de Blender

Vertex Paint
Ces Scripts sont disponibles dans le menu **Paint** de la **Vue 3D**.

FaceSelect
Ces Scripts sont disponibles dans le menu **Select** de la **Vue 3D** en mode **UV Face Select**.

WeightPaint
Ces Scripts sont disponibles dans le menu **Paint** de la **Vue 3D**.

UVCalculation
Ces Scripts sont disponibles comme options dans le menu **UV Calculation** ouvert avec **U** dans la **Vue 3D** en mode **UV Face Select**.

Script Templates
A Submenu called 'Script Templates' added to the text file menu, so that custom scripts can be added at those locations.

18.2. Insérer proprement un nouveau script dans les menus

Les créatteurs de scripts ne pensent pas toujours à doter leur travaux d'un **en-tête** permettant de les faire apparaître dans le menu **Scripts** de **Blender**. C'est le genre de chose qu'utilisateur de base peut réaliser lui-même, quoique avec prudence. En effet, il existe **deux types** de scripts. Le premier, le script simple (constitué d'un seul et unique fichier) est facilement modifiable en ajoutant quelques lignes au début de son texte. Le second (constitué d'un **nombre variable de modules** dans lesquels l'auteur a regroupé ses fonctions, et c'est pour cette raison qu'il ne possède pas d'en-tête pour uns menu) est plus délicat à traiter : vous ne pouvez pas le modifier et l'installer sans risque.

1/ En-tête pour un fichier simple
Voici un exemple d'en-tête pour un fichier simple :

```
#!BPY
"""
Name: 'Paths (.svg, .ps, .eps, .ai, Gimp)'
Blender: 233
Group: 'Import'
Submenu: 'Gimp 1.0 - 1.2.5' Gimp_1_0
Submenu: 'Gimp 2.0' Gimp_2_0
Submenu: 'Illustrator (.ai) PS-Adobe-2.0' AI
Submenu: 'InkScape (.svg)' SVG
Submenu: 'Postscript (.eps/.ps) PS-Adobe-2.0' EPS
Tip: 'Import a path from any of a set of formats (still experimental)'
"""
```

Tous les en-têtes des scripts que vous souhaitez voir apparaître dans les menus de **Blender** doivent commencer par les cinq caractères suivants : **#!BPY** .

Cette expression indique à l'API **Python** que ce qui se trouvera ensuite dans l'espace de texte défini par les triples guillemets (**"""**) doit être interprété comme une mise en forme pour la hiérarchie de menus.

- **Name:** **'**texte**'**
- **Blender:** numéro de version
- **Group:** **'**type**'**
- **Submenu:** **'**texte**'** argument
- **...**
- **Tip:** **'**texte**'**

Les lignes **Name**, **Blender**, **Group** et **Tip** sont obligatoires. Les guillemets simples autour des chaine de caractère également.

Le texte qui suit **Name** sera affiché dans le menu, mais n'a pas à respecter une forme particulière. Toutefois, il vaut mieux vérifier que ce nom n'est pas déjà utilisé par un autre script.

Le numéro de version qui suit **Blender** empêchera l'utilisation d'une version antérieure de **Blender** qui ne contiendrait pas certaines fonctions nécessaires de l'API.

Le type qui suit **Group** doit exister dans la version correspondante. Par défaut, et en cas de doute, il est préférable de choisir le groupe **Misc** qui correspond à **Divers** mais il faut éviter le groupe **Help** qui force l'affichage dans un autre menu que **Scripts**.

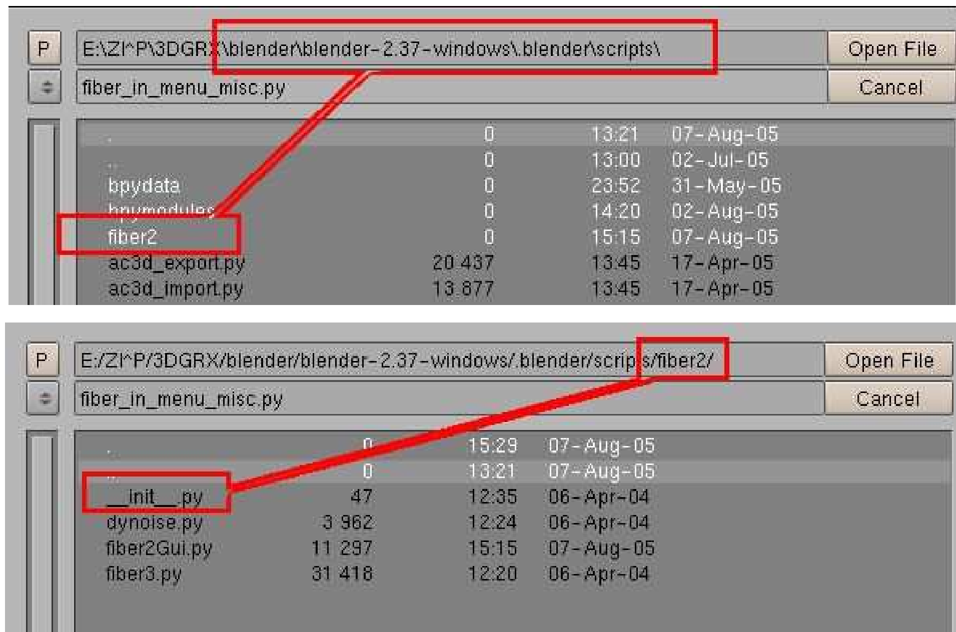
Submenu est optionnel est doit être suivi d'un premier texte libre qui apparaitra dans le menu et d'un second qui sera passé au script lui-même comme s'il s'agissait d'un argument en ligne de commande (voyez pour cela, la documentation du module **sys** et de la propriété **argv** en python).

Enfin, le script doit se trouver dans le répertoire **.blender/scripts** ou dans le répertoire qui a été défini comme **pythonpath** dans la fenêtre des préférences utilisateur (**Info**).

Attention : Si vous souhaitez modifier le **pythonpath**, il faut fournir une adresse valide : **f:\didacticiel\blender\tutor** ne passera pas alors que **f:\didacticiel\blender\tutor** ne posera pas de problème. Il est indispensable d'utiliser le bouton d'update pour reconstruire la liste.

2/ Le principe du site-package appliqué au menu Scripts
Certains scripts sont constitués de plusieurs modules. Les développeurs ayant l'habitude de nommer leurs bibliothèques de fonctions **math.py** ou **matrix.py**, si vous vous servez de la méthode précédente pour ajouter une entrée dans le menu, vous risquez d'écraser les fichiers qui portent déjà le même nom. La parade à ce type de problème est toute simple : créez dans le répertoire **.blender/scripts**, un sous-répertoire pour accueillir tous les modules et ajoutez-leur un fichier **__init__.py** qui contiendra un appel à tous les module nécessaires.

Par exemple, pour le script **fiber2**, cela donne :



et le fichier **__init__.py** contient la ligne : **import fiber2Gui**

Normalement, en ajoutant dans le répertoire **.blender/scripts**, le fichier **fiber_in_menu_misc.py** (le nom exact a peu d'importance, il doit simplement être différent de ceux qui s'y trouvent déjà) qui contient ces lignes :

```
#!BPY
""" Registration info for Blender menus: <- these words are ignored
Name: 'Fiber2'
Blender: 232
Group: 'Misc'
Tip: 'import fiber2 from site-packages.'
"""
import fiber2
```

vous devriez obtenir un résultat, mais ce n'est pas suffisant pour les menus de **Blender**. Il vous faut appeler directement le fichier **__init__** :

```
...
import fiber2.__init__
```

Ce qui revient à écrire :

```
...
import fiber2.fiber2Gui
```

3/ Le site-package ne permet pas d'utiliser le menu pour rappeler le même script

C'est un phénomène assez connu : le script finit de s'exécuter mais le python ne supprime pas complètement les modules de son espace de nom. Vous pouvez le contrôler facilement en utilisant le code suivant :

```
import sys
for m in sys.modules.keys():
 if m.find('fiber2')!=-1 : print m, sys.modules[m]
```

Vous obtenez l'affichage suivant dans la fenêtre **Console** :

```
fiber2.dynoise <module 'fiber2.dynoise' from 'E:\zi^p\3DGRX\blender\blender-2.37-windows\
.blender\scripts\fiber2\ dynoise.pyc'>
fiber2.fiber3 <module 'fiber2.fiber3' from 'E:\zi^p\3DGRX\blender\blender-2.37-
windows\.blender\scripts\fiber2\ fiber3.pyc'>
fiber2.math None
fiber2.fiber2Gui <module 'fiber2.fiber2Gui' from 'E:\zi^p\3DGRX\blender\blender-2.37-
windows\.blender\scripts\ fiber2\fiber2Gui.pyc'>
fiber2.string None
fiber2 <module 'fiber2' from 'E:\zi^p\3DGRX\blender\blender-2.37-windows\.blender\scripts\
fiber2\__init__.pyc'>
fiber2.sys None
fiber2.Blender None
fiber2.os None
```

La solution consiste à vérifier que ces noms de modules (les fonctions ont bien été nettoyées) ne sont plus dans ce dictionnaire et, si c'est nécessaire, à les supprimer dans le script d'appel :

```
#!BPY
""" Registration info for Blender menus: <- these words are ignored
Name: 'Fiber2'
Blender: 232
Group: 'Misc'
Tip: 'import fiber2 from site-packages.'
"""
import sys
for mod in sys.modules.keys():
 if mod.find('fiber2') !=-1 : del sys.modules[mod]
import fiber2.__init__
```

Essayez avec ce fichier par exemple :
http://cobalt3d.free.fr/didacticiel/blender/tutor/images/python/menu_packages/Fiberinmenuwizards.zip.

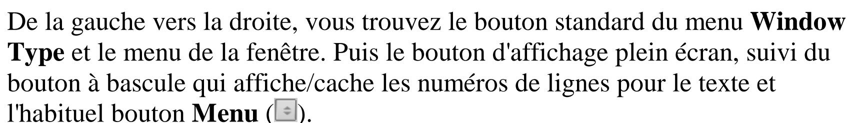
18.3. L'Ecriture de Scripts Python (Python Scripting)

Blender possède une fonctionnalité puissante encore très souvent négligée. Il propose en interne un puissant interpréteur **Python**. Ceci permet à tout utilisateur d'ajouter des fonctions en écrivants un scripts **Python**. **Python** est un langage de programmation orienté objet, interprété et interactif. Il incorpore des modules, des exceptions, une frappe de texte dynamique, des types de données dynamiques de très haut niveau et des classes. **Python** associe une puissance remarquable avec une syntaxe très claire. Il a été expressément conçu pour être utilisable comme langage d'extension pour les applications qui nécessitent une interface programmable, et c'est la raison pour laquelle il est utilisé par **Blender**.

Des deux méthodes principales pour étendre **Blender**, l'autre étant des plugins binaires, l'écriture de scripts **Python** est plus puissant, plus souple d'emploi, facile à comprendre et robuste. Il est généralement préférable d'utiliser un script **Python** plutôt que d'écrire un plugin.

Jusqu'à **Blender version 2.25**, dernière réalisation de **NaN**, l'écriture de scripts **Python** disposait de fonctionnalités quelque peu limitées. Depuis que **Blender** est Open Source, beaucoup des nouveaux développeurs réunis autour de la Fondation ont choisi de travailler dessus, de concert avec la modification de l'UI (Interface Utilisateur) et l'API **Python** est probablement la partie de **Blender** qui a obtenu le plus grand développement. Une réorganisation complète de ce qui existait a été conduite et de nombreux nouveaux modules ont été ajoutés. Cette évolution est toujours en cours et une intégration encore meilleure est espérée dans les versions à venir de **Blender**.

Blender possède un éditeur **Text** parmi les types de fenêtres accessibles via le bouton Text Editor du menu **Window Type**, ou par **SHIFT F11**.
La fenêtre de l'éditeur **Text** nouvellement ouverte est grise et vide, avec une barre d'outils très simple (image ci-contre).



De la gauche vers la droite, vous trouvez le bouton standard du menu **Window Type** et le menu de la fenêtre. Puis le bouton d'affichage plein écran, suivi du bouton à bascule qui affiche/cache les numéros de lignes pour le texte et l'habituel bouton **Menu** ( ).

Le bouton **Menu** vous permet de sélectionner quel tampon texte est à afficher, de même qu'il vous permet de créer un nouveau tampon ou de charger un fichier texte.

Si vous choisissez de charger un fichier, l'éditeur **Text** devient temporairement une fenêtre de sélection de fichiers, avec les fonctions habituelles. Une fois qu'un tampon texte est chargé dans l'éditeur **Text**, celle-ci se comporte fonctionne comme un éditeur de texte très simple. Taper sur les touches du clavier produit un texte dans le tampon texte. Comme d'habitude, presser **LMB** drague et relâcher **LBM** sélectionne du texte.

Les commandes claviers suivantes s'appliquent :

- **ALT C** ou **CTRL C** : Copie le texte sélectionné dans le presse-papiers texte;
- **ALT X** ou **CTRL X** : Coupe le texte sélectionné et l'envoie dans le presse-papiers texte ;
- **ALT V** ou **CTRL V** : Colle le texte depuis le presse-papiers texte à l'emplacement du curseur dans la fenêtre **Text** ;
- **ALT S** : Sauvegarde le texte dans un fichier texte, une fenêtre de sélection de fichiers apparaît ;
- **ALT O** : Charge un texte, une fenêtre de sélection de fichiers apparaît ;
- **ALT F** : Fait apparaître la boîte à outils **Find** ;
- **SHIFT ALT F** ou **RMB** - Fait apparaître le menu **File** pour la fenêtre **Text** ;
- **ALT J** : Fait apparaître un bouton numérique où vous pouvez spécifier un n° de ligne où le curseur sautera (jump) ;
- **ALT P** : Exécute le texte en tant que script **Python** ;
- **CRTL Z** : Undo ;
- **CTRL SHIFT Z** : Redo ;
- **ALT R** : Recharge le tampon courant ;
- **ALT M** : Convertit le contenu de l'éditeur **Text** en texte 3D (maximum 100 caractères) ;

Le presse-papiers de **Blender** pour le couper/copier/coller est distinct du presse-papiers de **Windows**. Donc, vous ne pouvez pas couper/copier/coller depuis/dans **Blender**. Pour accéder à votre presse-papiers **Windows**, utilisez **SHIFT CTRL C** (copier) et **SHIFT CTRL V** (coller).

Pour effacer un tampon texte, cliquez simplement le bouton **X** à côté du nom du tampon, exactement comme vous le feriez pour un matériau, etc...

Le plus important des raccourcis est **ALT P** qui fait que le contenu du tampon est analysé par l'interpréteur **Python** interne, intégré à **Blender**.

Le paragraphe suivant vous présentera un exemple d'écriture de script **Python**. Avant de commencer, il est important de noter que **Blender** ne contient que l'interpréteur **Python** de base et avec un nombre restreint de modules spécifiques à **Blender** et décrits dans les références.

Pour accéder aux modules standard de **Python**, vous devez avoir installé sur votre machine un **Python** complet et opérationnel. Vous pouvez le télécharger depuis **http://www.python.org**. Mais vérifiez sur **http://www.blender.org** quelle est la version exacte de **Python** qui a été intégrée à **Blender** de façon à rester compatible.

> **Conseil Pratique** : La fenêtre de l'éditeur **Text** est aussi très pratique quand vous voulez partager vos fichiers **.blend** avec la communauté ou avec vos amis. L'éditeur **Text** peut être utilisée pour écrire un fichier **README** interne expliquant le contenu de votre fichier **Blender**. Bien plus pratique que de le faire dans une application indépendante. Assurez-vous qu'elle est visible quand vous sauvegardez ! Si vous partagez le fichier avec la communauté, et que vous voulez le partager sous une licence quelconque, vous pouvez écrire la licence dans l'éditeur **Text**.

18.2.1. Régler la variable d'environnement PYTHONPATH

(Setting the PYTHONPATH environment variable)

Blender doit aussi être averti de l'endroit où se trouve l'installation complète de **Python**. Ceci est fait en utilisant la variable d'environnement **PYTHONPATH**.

Réglage de PYTHONPATH sous Windows

Une fois que vous avez installé **Python** (actuellement en version **2.5**), disons dans **C:\PYTHON25**. Cliquez avec **RMB** sur l'icône **Poste de Travail** sur le bureau, sélectionnez **Propriétés**. Puis l'onglet **Avancé** et cliquez sur le bouton **Variables d'environnement**.

En-dessous de la boîte **Variables Système**, cliquez sur **Nouveau**. Si vous n'êtes pas administrateur, vous n'aurez pas le droit de faire ça. Dans ce cas, cliquez sur **Nouveau** dans la boîte supérieure.

Maintenant, dans la boîte **Nom de la variable**, entrez **PYTHONPATH**, dans la boîte **Valeur de la variable**, entrez :

C:\PYTHON25;C:\PYTHON25\DLLS;C:\PYTHON25\LIB;C:\PYTHON25\LIB\LIB-TK

Appuyez sur **OK** autant de fois que nécessaire pour sortir de toutes les boîtes de dialogue. Vous devrez (ou ne devrez pas) rebooter, selon votre **OS**.

18.2.2. Un Exemple Fonctionnel en Python (A working Python example)

Maintenant que vous avez vu que **Blender** est extensible via des scripts **Python** et que avez acquis les bases de l'écriture des scripts et sur la façon de lancer un scripts, avant de vous compliquer l'esprit avec les références complètes de l'API **Python**, jetez un coup d'oeil à un exemple fonctionnel simple.
Vous allez étudier un petit script pour produire des polygones. C'est une sorte de clône du menu **Add > Mesh > Circle**, mais qui crée des polygones pleins, et pas seulement leur contour.
Pour rendre ce script complet, bien que simple, il fournira une **GUI** (Graphical User Interface = Interface Graphique Utilisateur) entièrement écrite avec l'API **Blender**.

Entête et Importation des Modules et des Valeurs Globales (Headers, importing modules and globals)
Les 32 premières lignes de code sont listées dans ci-dessous :

```
001 ######################################################
002 #
003 # Script de Démo pour Blender 2.3
004 #
005 ###################################################S68
006 # Ce script génére des polygones. Il n'est pas très utile puisque
007 # vous pouvez créer des polygones avec ADD>Mesh>Circle
008 # mais c'est un bel exemple de script complet, et les
009 # polygones sont 'pleins'
010 ######################################################
011
012 ######################################################
013 # Importation des modules
014 ######################################################
015
016 import Blender
017 from Blender import NMesh
018 from Blender.BGL import *
019 from Blender.Draw import *
020
021 import math
022 from math import *
023
024 # Paramètres d'un Polygone
025 T_NumberOfSides = Create(3)
026 T_Radius = Create(1.0)
027
028 # Evénements
029 EVENT_NOEVENT = 1
030 EVENT_DRAW = 2
031 EVENT_EXIT = 3
032
```

Après les commentaires nécessaires avec la description de ce que fait le script, il y a (lignes 016 à 022) l'importation de modules **Python**.

Blender est le module principal de l'API **Python** de **Blender**. **NMesh** est le module fournissant un accès aux Maillages de **Blender**, tandis que **BGL** et **Draw** donnent respectivement accès aux constantes et fonctions **OpenGL** et à l'interface à fenêtres de **Blender**. Le module **math** est le module des fonctions mathématiques de **Python**, mais comme les modules **math** et **os** sont intégrés dans **Blender**, vous n'avez pas besoin d'une installation complète de **Python** pour cela!

Les polygones sont définis par leur nombre de côtés et leur rayon. Ces paramètres ont des valeurs qui doivent être définies par l'utilisateur via l'interface graphique (**GUI**), donc les lignes 025 à 026 créent deux objets 'boutons génériques', avec leur valeur de départ par défaut.
Finalement, les Objets **GUI** travaillent avec des événements (Events) et en génèrent. Les identificateurs d'événements sont des entiers que le programmeur doit définir. C'est habituellement une bonne pratique que de donner des noms mnémotechniques aux évènements, comme dans les lignes 029 à 031.

Dessin de l'Interface Graphique (Drawing the GUI)

Le code responsable de la construction de l'interface graphique (**GUI**) doit être contenu dans une fonction **draw** :

```
033 ######################################################
034 # Dessin de la GUI
035 ######################################################
036 def draw():
037         global T_NumberOfSides
038         global T_Radius
039         global EVENT_NOEVENT,EVENT_DRAW,EVENT_EXIT
040
041         ########## Titres
042         glClear(GL_COLOR_BUFFER_BIT)
043         glRasterPos2d(8, 103)
044         Text("Demo Polygon Script")
045
046         ######### Paramètres des boutons de la GUI
047         glRasterPos2d(8, 83)
048         Text("Parameters:")
049         T_NumberOfSides = Number("No. of sides: ", EVENT_NOEVENT, 10, 55, 210, 18,
050                 T_NumberOfSides.val, 3, 20, "Number of sides of out polygon");
051         T_Radius = Slider("Radius: ", EVENT_NOEVENT, 10, 35, 210, 18,
052                 T_Radius.val, 0.001, 20.0, 1, "Radius of the polygon");
053
054         ######### Boutons Draw et Exit
055         Button("Draw",EVENT_DRAW , 10, 10, 80, 18)
056         Button("Exit",EVENT_EXIT , 140, 10, 80, 18)
057
```

Les lignes 037 à 039 donnent simplement accès aux données globales. Le code principal commence réellement aux lignes 042 à 044. La fenêtre **OpenGL** est initialisée, et sa position courante est fixée en X = 8 et Y = 103. L'origine de cette référence (0,0) est le coin inférieur gauche de la fenêtre du script. Puis, le titre **Demo Polygon Script** est affiché.

Une autre chaîne de caractères est affichée (lignes 047 à 048), puis les boutons d'entrée pour les paramètres sont créés. Le premier (lignes 049 à 050) est un bouton numérique, exactement comme ceux que vous trouvez dans les divers Contextes de **Blender**. Pour la signification de tous les paramètres, référez-vous aux références de l'API. Basiquement, vous trouvez le label du bouton, l'événement qu'il génère, sa position (X,Y) et ses dimensions (largeur, hauteur), sa valeur, qui est une donnée qui appartient à l'Objet **Button** lui-même, les valeurs minimale et maximale permises et une chaîne texte qui apparaîtra comme une aide lorsque la souris le survolera (Tooltips).

Les lignes 051 à 052 définissent un bouton numérique avec curseur, avec une syntaxe très similaire. Les lignes 055 à 056 créent finalement un bouton **Draw** qui produirra le polygone et un bouton **Exit** pour quitter le script.

Gestion des Evénements (Managing Events)

L'interface graphique (**GUI**) n'est pas dessinée et ne fonctionnera pas, tant qu'un gestionnaire approprié d'évènements n'aura pas été écrit et enregistré :

```
058 def event(evt, val):
059         if (evt == QKEY and not val):
060                 Exit()
061
062 def bevent(evt):
063         global T_NumberOfSides
064         global T_Radius
065         global EVENT_NOEVENT,EVENT_DRAW,EVENT_EXIT
066
067         ######### Gestion des évènements GUI
068         if (evt == EVENT_EXIT):
069                 Exit()
070         elif (evt== EVENT_DRAW):
071                 Polygon(T_NumberOfSides.val, T_Radius.val)
072                 Blender.Redraw()
073
074 Register(draw, event, bevent)
075
```

Les lignes 058 à 060 définissent le gestionnaire des événements claviers, ici la réponse à **Q** avec un appel à **Exit()**.

Plus intéressantes sont les lignes 062 à 072, qui prennent en charge la gestion des événements de la GUI. Chaque fois qu'un bouton de l'interface graphique (**GUI**) est utilisé, cette fonction est appelée, avec le numéro d'événement défini comme paramètre dans le bouton. Le coeur de cette fonction est une structure **Select** (sélectionner) qui exécute différentes parties du code en fonction du numéro d'événement.

Comme dernier appel, la fonction **Register** est appelée. Elle dessine effectivement l'interface graphique (**GUI**) et démarre le cycle de capture des événements.

Gestion du Maillage (**Mesh handling**)

Finalement, la partie ci-dessous montre la fonction principale, celle qui va créer le polygone. C'est une édition de Maillage assez simple, mais elle montre plusieurs points importants de la structure de données interne de **Blender**.

```
076 ######################################################
077 # Corps Principal
078 ######################################################
079 def Polygon(NumberOfSides,Radius):
080
081         ######### Crée un nouveau Maillage
082         poly = NMesh.GetRaw()
083
084         ######### Remplissage avec des vertices
085         for i in range(0,NumberOfSides):
086                 phi = 3.141592653589 * 2 * i / NumberOfSides
087                 x = Radius * cos(phi)
088                 y = Radius * sin(phi)
089                 z = 0
090
091                v = NMesh.Vert(x,y,z)
092                 poly.verts.append(v)
093         ######### Ajout d'un nouveau vertex au centre
094         v = NMesh.Vert(0.,0.,0.)
095         poly.verts.append(v)
096
097         ######### Connecte les vertices pour former des faces
098         for i in range(0,NumberOfSides):
099                 f = NMesh.Face()
100                 f.v.append(poly.verts[i])
101                 f.v.append(poly.verts[(i+1)%NumberOfSides])
102                 f.v.append(poly.verts[NumberOfSides])
103                 poly.faces.append(f)
104
105         ######### Crée un nouvel Objet avec le nouveau Maillage
106         polyObj = NMesh.PutRaw(poly)
107
108         Blender.Redraw()
109
```

La première ligne importante est la ligne 082. Ici, un nouvel objet **Maillage**, **poly** est créé. L'objet **Maillage** est constitué d'une liste de vertices et d'une liste de faces, plus quelques autres choses intéressantes. Dans votre exemple, les vertices et les faces sont ce dont vous avez besoin.

Bien sûr, le nouveau Maillage créé est vide. La première boucle (lignes 085 à 092) calcule la position X,Y,Z des **NumberOfSides** vertices nécessaires pour définir le polygone. La figure étant plane, c'est Z = 0 pour tous.

La ligne 091 appelle la méthode **Vert** de **NMesh** pour créer un nouvel Objet **Vertex** de coordonnées (X,Y,Z). Cet Objet est ensuite ajouté (ligne 096) à la liste **verts** du Maillage **poly**.

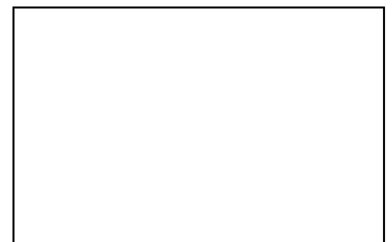
Finalement, (lignes 095 à 096) un dernier vertex est ajouté au centre.

Les lignes 099 à 104 connectent maintenant ces vertices pour créer des faces. Il n'est pas obligatoire de créer tous les vertices à l'avance et ensuite les faces. Vous pouvez créer sans problème une nouvelle face dès que tous ses vertices sont présents.

La ligne 100 crée un nouvel Objet **Face**. Un objet **Face** possède sa propre liste de vertices **v** (jusqu'à 4) pour le définir. Les lignes 101 à 103 ajoutent 3 vertices à la liste **f.v** vide à l'origine. Les vertices sont deux vertices consécutifs du polygone et le vertex central.

Ces vertices doivent être pris dans la liste **verts** du Maillage. Finalement, la ligne 104 ajoute la face nouvellement créée à la liste **faces** de votre Maillage **poly**.

Conclusion



Si vous créez un fichier **polygon.py** contenant le code décrit ci-dessus, que vous le chargez dans une fenêtre **Text** de **Blender**, comme vous l'avez appris dans le paragraphe précédent, et que vous appuyez sur **ALT P** dans cette fenêtre pour le lancer, vous verrez disparaître le script et la fenêtre deviendra grise. L'interface graphique (**GUI**) s'affichera dans le coin inférieur gauche (image ci-contre).

En sélectionnant, par exemple, 5 vertices et un rayon de 0.5, et en cliquant sur le bouton **Draw**, un pentagone apparaîtra sur le plan XY de la **Vue 3D** (image ci-contre).

Un exemple fonctionnel : Le script **Weight Gradient…** (weightpaint_gradient.py) dans le menu **Paint** du mode **Weight Gradient** :

```
#!BPY
"""
Name: 'Weight Gradient...'
Blender: 241
Group: 'WeightPaint'
Tooltip: 'Click on the start and end grad points for the mesh for selected faces.'
"""

__author__ = ["Campbell Barton"]
__url__ = ("blender", "elysiun", "http://members.iinet.net.au/~cpbarton/ideasman/")
__version__ = "0.1"
__bpydoc__=\

'''
This script is used to fill the selected faces with a gradient
To use the script, switch to "Face Select" mode then "Vertex Paint" mode
Select the faces you wish to apply the gradient to.
Click twice on the mesh to set the start and end points of the gradient.
The color under the mouse will be used for the start and end blend colors.
Note:
Holding Shift or clicking outside the mesh on the second click will blend the first colour to nothing.
'''

import mesh_gradient
import Blender
def main():
	scn= Blender.Scene.GetCurrent()
	ob= scn.getActiveObject()
	if not ob or ob.getType() != 'Mesh':
		Blender.Draw.PupMenu('Error, no active mesh object, aborting.')
		return
	# MODE 0 == VCOL
	# MODE 1 == WEIGHT
	MODE= 0
	mesh_gradient.vertexGradientPick(ob, MODE)

if __name__ == '__main__':
	main()
```

18.2.3. La Documentation de Référence pour Python

L'Interface complète de Programmation d'une Application en Python (**API** = Application Programmer Interface) sous **Blender** possède une documentation de référence qui occupe un livre à elle seule. Pour une raison de place, elle n'est pas incluse ici.

Vous pouvez trouver cette **API Python** pour **Blender 2.42** à l'adresse suivante :
http://www.blender.org/modules/documentation/242PythonDoc/ .

18.2.4. Les Scripts en Python

Il existe, sur **Internet**, plus d'une centaine de scripts différents disponibles pour **Blender**. Comme pour les plugins, les scripts sont très dynamiques, et ils changent d'interface, de fonctionnalités et d'emplacements sur Internet assez rapidement, aussi pour trouver une liste mise à jour et des liens fonctionnels, référez-vous à l'un des deux principaux sites pour **Blender** : **http://www.blender.org** ou **http://www.elysiun.com**.

18.3. Le Système de Plugins de Blender (Blender's Plugins System)

Ce paragraphe explique en profondeur comment coder des plugins de Texture ou de Séquence pour **Blender**.

18.3.1. Ecrire un Plugin de Texture (Writing a Texture Plugin)

Dans ce paragraphe, vous allez écrire un plugin de Texture basique et ensuite passer en revue les différentes étapes de son utilisation. Les bases derrière un plugin de Texture est qu'il vous est demandé quelques entrées : position et valeur normales, ainsi que quelques autres informations. Puis, vous retournez l'intensité, la couleur et/ou l'information normale en fonction du type de plugin de Texture.

Tous les fichiers nécessaires au développement des plugins ainsi que quelques exemples de plugins se trouvent dans le répertoire **blender/plugins**. Vous pouvez alternativement obtenir une série de plugins à l'adresse **http://www.cs.umn.edu/~mein/blender/plugins**.

Les plugins sont supportés (chargés/appelés) dans **Blender** en utilisant la famille d'appels **dlopen()**. Pour ceux qui ne connaissent pas le système **dlopen()**, sachez qu'il permet à un programme (**Blender**) d'utiliser un Objet compilé comme s'il faisait partie du programme lui-même, similaire aux bibliothèques liées dynamiquement, sauf que les Objets à charger sont déterminés en temps réel.

L'avantage de l'utilisation du système **dlopen()** pour les plugins est qu'il est très rapide pour accéder à une fonction, et qu'il n'y a pas de délais dans l'interfaçage avec le plugin, ce qui est déterminant quand (cas des plugins de Texture), un plugin peut être appelé plusieurs millions de fois pendant un unique rendu.

Le désavantage du système est que le code du plugin fonctionne comme s'il était une partie de **Blender**, si le plugin plante, **Blender** plante aussi.

Les fichiers **include** trouvés dans le sous-répertoire **plugin/include/** de l'installation de **Blender** documentent les fonctions de **Blender** destinées aux plugins. Ceci inclut les fonctions de la bibliothèque **Imbuf** pour charger et travailler avec des images et des tampons d'images, ainsi que les fonctions **noise** et **turbulence** pour un texturage cohérent.

Les Spécifications

#include <plugin.h>

Chaque plugin de **Blender** doit inclure ce fichier d'en-tête, qui contient toutes les structures et définitions nécessaires pour travailler correctement avec **Blender**.

char name[]="Tiles";

Une chaîne de caractères contenant le nom du plugin, cette valeur sera affichée comme titre de la texture dans le panneau **Texture**.

define NR_TYPES 2 char stnames[NR_TYPES][16]= {"Square", "Deformed"};

Les plugins ont la possibilité d'avoir des sous-types séparés pour des variations minimes des algorithmes - par exemple, la texture par défaut **Cloud** dispose des sous-types **Default** et **Color**. **NR_STYPES** doit être défini comme étant le nombre de sous-types nécessaires pour votre plugin, et un nom doit être attribué à chaque sous-type. Chaque plugin doit avoir au moins 1 sous-type et un nom de sous-type.

VarStruct varstr[]= {...};

varstr contient toutes les informations dont **Blender** a besoin pour afficher des boutons pour un plugin. Les boutons de plugin peuvent être numériques pour l'entrée de données, ou texte pour des commentaires et d'autres informations. Les plugins sont limité à un maximum de 32 variables. Chaque entrée **VarStruct** est constituée d'un type, d'un nom, d'une information d'intervalle (range) et d'une note d'information (Tooltip) :

- Le **type** définit le type de données pour chaque entrée de bouton, et la façon d'afficher le bouton. Pour les boutons numériques, cette valeur est une combinaison (**OR** = OU logique) de **INT** (entier) ou de **FLO** (virgule flottante) pour le format du nombre, et de **NUM** (numérique), de **NUMSLI** (numérique à curseur) ou de **TOG** (à bascule 2 états) pour le type du bouton. Les boutons Texte doivent avoir le type **LABEL**.
- Le **nom** est ce qui sera affiché sur le (ou à côté du) bouton. Ceci est limité à 15 caractères.
- L'information d'intervalle est constituée de trois nombres en virgule flottante qui définissent respectivement la valeur par défaut, la valeur minimum et la valeur maximum du bouton. Pour les boutons **TOG**, le minimum est réglé dans l'état appuyé et le maximum est réglé dans l'état relâché.
- La note d'information (ou tooltip) est une chaîne qui sera affichée quand la souris est au-dessus de ce bouton (si l'utilisateur a validé la fonction **Tooltips**). Ceci est limité à 80 caractères, et doit être réglé sur une chaîne **NULL** ("") si elle n'est pas utilisée.

typedef struct Cast {...};

La structure **Cast** est utilisée dans l'appel de la fonction **doit**, et sert de moyen pour accéder simplement aux valeurs des données de chaque plugin. **Cast** doit contenir, dans l'ordre, un entier ou un nombre en virgule flottante pour chaque bouton

défini dans **varstr**, en incluant les boutons texte. Typiquement, ceux-ci doivent avoir le même nom que le bouton à référence simple.

float result[8];
Le tableau **résult** est utilisé pour transmettre des informations depuis le plugin et en recevoir de ce dernier. Les valeurs **result** sont présentées ci- dessous :

Indice de result	Signification	Intervalle
result[0]	Valeur **Intensity**	0.0 à 1.0
result[1]	Valeur **Color** Rouge (**R**)	0.0 à 1.0
result[2]	Valeur **Color** Verte (**G**)	0.0 à 1.0
result[3]	Valeur **Color** Bleue (**B**)	0.0 à 1.0
result[4]	Valeur **Color** Alpha	0.0 à 1.0
result[5]	Valeur **Normal** - Déplacement en X	-1.0 à 1.0
result[6]	Valeur **Normal** - Déplacement en Y	-1.0 à 1.0
result[7]	Valeur **Normal** - Déplacement en Z	-1.0 à 1.0

Le plugin doit toujours retourner une valeur **Intensity**. Retourner une valeur **Color** (RGB) ou une valeur **Normal** (déplacement de normale) est optionnel, et doit être indiqué par le drapeau **return** de la fonction **doit** réglé à 1 (**Color**) ou à 2 (**Normal**). Avant l'appel du plugin, **Blender** inclut le rendu courant des Normales dans **result[5]**, **result[6]** et **result[7]**.

float cfra
La valeur **cfra** est réglée par **Blender** à la valeur courante avant chaque passe de rendu. Cette valeur est un nombre
De cellos +/- .5 en fonction du réglage des champs.

plugin_tex_doit prototype
La fonction **plugin_tex_doit** doit être prototypée pour être utilisée par la fonction **getinfo**. Vous n'avez pas à modifier cette ligne.

plugin_tex_getversion
Cette fonction doit exister dans chaque plugin pour qu'il soit correctement chargé. Vous n'avez pas à modifier cette fonction.

plugin_but_changed
Cette fonction est utilisée pour transmettre de l'information sur les boutons qui ont été modifiés par l'utilisateur dans l'interface. La plupart des plugins n'ont pas besoin d'utiliser cette fonction, sauft quand l'interface permet à l'utilisateur de modifier une variable quelconque qui oblige le plugin à recalculer (par exemple, une table de nombres aléatoires).

plugin_init
Si nécessaire, des plugins peuvent utiliser cette fonction pour initialiser des données internes. **Note** : Cette fonction **init** peut être appelée de nombreuses fois si le même plugin de Texture est copié. N'initialisez pas des données globales spécifiques pour une seule instance d'un plugin dans cette fonction.

plugin_getinfo
Cette fonction est utilisée pour transmettre des informations à **Blender**. Vous n'avez jamais besoin de la modifier.

plugin_tex_doit
La fonction **doit** est responsable du retour d'informations vers **Blender** à propos du pixel demandé.

Les Arguments

int stype : C'est le nombre de sous-types sélectionnés, voir ci-dessus les entrées **NR_TYPES** et **char stypes**.

Cast *cast : La structure **Cast** qui contient les données du plugin, voir ci-dessus l'entrée **Cast**.

float *texvec : Ceci est un pointeur vers 3 nombres en virgule flottante, qui sont les coordonnées de texture pour lesquelles une valeur de texture a été retournée.

float *dxt float *dyt : Si ces pointeurs sont non-**NULL**, ils pointent vers deux vecteurs (deux tableaux de trois nombres en virgule flottante) qui définissent la valeur de la taille de la texture considérée en pixels. Ils sont seulement non-**NULL** quand **OSA** est validé et sont utilisés pour calculer correctement l'anti-crénelage (anti-aliasing).
La fonction **doit** doit remplir le tableau **return** et renvoyer 0, 1, 2 ou 3 en fonction des valeurs qui y ont été inscrites. La fonction **doit** doit **toujours** remplir une valeur **Intensity**. Si la fonction remplit une valeur **Color**, elle doit renvoyer 1, 2 pour une valeur **Normal**, et 3 pour toutes les autres valeurs.

Interaction Texture/Matériau

Blender est un peu différent de la plupart des logiciels 3D du fait de la séparation logique des Textures et des Matériaux. Dans **Blender**, les Textures sont des Objets qui renvoient certaines valeurs, en fait, des générateurs de signaux. Les Matériaux contrôlent le mapping des Textures sur les objets, ce qui est affecté, avec quelle intensité, de quelle manière, etc...
Des plugins correctement conçus doivent seulement inclure des variables pour modifier le signal renvoyé, et pas son mapping. Des boutons pour contrôler l'échelle (scale), l'intervalle (range), les axes (axis), etc... sont parfaits uniquement s'ils servent à rendre plus facile l'utilisation du plugin (cas du bouton **Size** du plugin **Tiles**) ou s'ils accélèrent les calculs (les sous-types **Intensity**/**Color**/**Bump** du plugin **Clouds2**). Sinon, les boutons du contexte **Material** rendent redondants ces boutons et l'interface devient inutilement compliquée.

Un Plugin de Texture Générique (Generic Texture Plugin)

```
#include "plugin.h"

/* Nom de la Texture */
char name[24]= "";

#define NR_TYPES 3
char stnames[NR_TYPES][16]= {"Intens","Color", "Bump"};

/* Structure pour les boutons,
 * butcode name default min max 0
 */
VarStruct varstr[]= {
        { NUM|FLO, "Const 1", 1.7, -1.0, 1.0, ""},
};

typedef struct Cast {
        float a;
} Cast;

float result[8];
float cfra;

int plugin_tex_doit(int, Cast*, float*, float*, float*);

/* Fonctions Fixées*/
int plugin_tex_getversion(void) {
        return B_PLUGIN_VERSION;
}

void plugin_but_changed(int but) { }
void plugin_init(void) { }

void plugin_getinfo(PluginInfo *info) {
        info->name= name;
        info->stypes= NR_TYPES;
        info->nvars= sizeof(varstr)/sizeof(VarStruct);
        info->snames= stnames[0];
        info->result= result;
        info->cfra= &cfra;
        info->varstr= varstr;
        info->init= plugin_init;
        info->tex_doit= (TexDoit) plugin_tex_doit;
        info->callback= plugin_but_changed;
}

int plugin_tex_doit(int stype, Cast *cast, float *texvec, float *dxt, float *dyt) {
        if (stype == 1) {
                return 1;
        } if (stype == 2) {
                return 2;
        }
        return 0;
}
```

Vos Modifications

La première étape consiste à avoir prévu un plan d'action. Quel est le but de ce plugin, que fait-il et comment les utilisateurs pourront-ils l'utiliser. Dans cet exemple, vous allez créer une Texture simple qui produira un motif simple de briques/blocs.
Maintenant, vous allez copier votre plugin générique dans le fichier **cube.c**, puis vous allez remplir les parties laissées vides.
C'est toujours une bonne idée d'ajouter quelques commentaires. Le premier indiquera aux utilisateurs ce que fait le plugin, où ils peuvent en obtenir une copie, qui ils doivent contacter en cas de bugs ou de dysfonctionnements, et quelles sont les restrictions de licence sur le code.
N'oubliez pas les /* */ pour encadrer ces commentaires. Les plugins sont écrits en langage **C** et certains compilateurs **C** n'acceptent pas les // pour encadrer les commentaires.

```
/*
Description : Ce plugin est un exemple de plugin de Texture qui crée un motif simple de briques/blocs.
Il demande deux valeurs une taille de brique et une taille de mortier.
La taille de la brique (brick size) est la taille de chaque brique.
La taille du mortier (mortar size) est la largeur du mortier entre les briques.
Auteur : Kent Mein (mein@cs.umn.edu)
Site Internet : http://www.cs.umn.edu/~mein/blender/plugins
Licence : Domaine Public
Dernière Modification : Jeudi 21 octobre 2003 05:57:13
*/
```

Ensuite, vous devez remplir le nom (**name**). Vous devriez conserver le même nom que celui de votre fichier **.c**, de préférence descriptif, avec un nombre de caractères inférieur à 23, sans espace et entièrement en minuscules.

```
char name[24]= "cube.c";
```

Vous voulez que ce plugin reste simple, et qu'il n'ai qu'un seul type qui traite de l'intensité (**Intensity**). Donc, vous avez besoin de ce qui suit :

```
#define NR_TYPES        1
char stnames[NR_TYPES][16]= {"Default"};
```

Pour votre interface utilisateur, vous allez permettre à l'utilisateur de pouvoir modifier la taille d'une brique et du mortier, ainsi que les valeurs de l'intensité renvoyées par la brique et le mortier. Pour cela, vous devez éditer **varstr** et **Cast**. **Cast** doit avoir une variable pour chaque entrée de **varstr**.

```
/* Structure pour les boutons,
*  butcode      name           default  min  max  Tool tip
*/
VarStruct varstr[]= {
        {NUM|FLO,   "Brick",      .8,    0.1,   1.0, "Size of Cell"},
        {NUM|FLO,   "Mortar",     .1,    0.0,   0.4, "Size of boarder in cell"},
        {NUM|FLO,   "Brick Int",   1,    0.0,   1.0, "Color of Brick"},
        {NUM|FLO,   "Mortar Int",  0,    0.0,   1.0, "Color of Mortar"},
};

typedef struct Cast {
        float brick,mortar, bricki, mortari;
} Cast;
```

Maintenant, vous devez remplir **plugin_tex_doit**. Vous voulez à la base séparer votre Texture en cellules qui seront constituées d'une brique et du mortier le long de la base de cette brique. Puis, vous voulez déterminer si vous êtes dans la brique ou dans le mortier. Le code suivant devrait réaliser cela :

```
int plugin_tex_doit(int stype, Cast *cast, float *texvec, float *dxt, float *dyt) {
        int c[3];
        float pos[3], cube;

        /* réglage de la taille de notre cellule */
        cube = cast->brick + cast->mortar;

        /* nous avons besoin de déterminer si nous sommes à l'intérieur de la brique actuelle. */
        c[0] = (int)(texvec[0] / cube);
        c[1] = (int)(texvec[1] / cube);
        c[2] = (int)(texvec[2] / cube);
        pos[0] = ABS(texvec[0] - (c[0] * cube));
        pos[1] = ABS(texvec[1] - (c[1] * cube));
        pos[2] = ABS(texvec[2] - (c[2] * cube));

        /* Calcule si nous sommes dans une position de mortier dans la brique ou pas. */
        if ((pos[0] <= cast->mortar) || (pos[1] <= cast->mortar) ||
            (pos[2] <= cast->mortar)) {
                result[0] = cast->mortari;
        } else {
                result[0] = cast->bricki;
        }
        return 0;
}
```

Une chose à noter, la fonction **ABS** est définie dans un en-tête placé dans le répertoire **plugins/include**. Vous y trouverez quelques autres fonctions communes : n'oubliez pas d'aller y jeter un coup d'œil.

Compilation

bmake est un utilitaire simple (un script shell) pour vous aider dans la compilation et le développement de plugins, et il peut être trouvé dans le sous-répertoire **plugins** du répertoire d'installation de **Blender**. Il est invoqué par : **bmake (nom_du_plugin.c)** et il essayera de lier les bibliothèques convenables et compilera correctement le fichier C spécifié pour votre système.

Si vous essayez de développer des plugins sur une machine **Windows**, **bmake** ne fonctionnera pas et vous devrez utiliser **lcc**. Pour compiler un plugin avec **lcc**, vous pouvez utiliser la méthode suivante après avoir vérifié que vous avez vos plugins dans le répertoire **c:\blender\plugins**. Vous avez ici un exemple vous montrant comment compiler le plugin de Texture **sinus.c**. Ouvrez une console **Dos** et exécutez ce qui suit :

(**Note** : Vérifiez bien auparavant que le répertoire **lcc\bin** apparaît bien dans votre variable d'environnement **PATH**).

```
cd c:\blender\plugins\texture\sinus
lcc -Ic:\blender\plugins\include sinus.c
lcclnk -DLL sinus.obj c:\blender\plugins\include\tex.def
implib sinus.dll
```

19.3.2. Ecrire un Plugin Sequence (Writing a Sequence Plugin)

Dans ce paragraphe, vous allez écrire un plugin **Sequence** basique et ensuite passer en revue les différentes étapes de son utilisation. Les bases derrière un plugin **Sequence** est qu'il vous est demandé quelques entrées : 1 à 3 tampons d'images en entrée ainsi que quelques autres informations. Puis, vous retournez le tampon de l'image résultante.
Tous les fichiers nécessaires au développement des plugins ainsi que quelques exemples de plugins se trouvent dans le répertoire **blender/plugins**. Vous pouvez alternativement obtenir une série de plugins à l'adresse **http://www.cs.umn.edu/~mein/blender/plugins**.

Les Spécifications

#include <plugin.h>
Chaque plugin de **Blender** doit inclure ce fichier d'en-tête, qui contient toutes les structures et définitions nécessaires pour travailler correctement avec **Blender**.

char name[]="Blur";
Une chaîne de caractères contenant le nom du plugin, cette valeur sera affichée comme titre de la séquence dans l'éditeur **Sequence**.

VarStruct varstr[]= {...};
varstr contient toutes les informations dont **Blender** a besoin pour afficher des boutons pour un plugin. Les boutons de plugin peuvent être numériques pour l'entrée de données, ou texte pour des commentaires et d'autres informations. Les plugins sont limité à un maximum de 32 variables. Chaque entrée **VarStruct** est constituée d'un type, d'un nom, d'une information d'intervalle (range) et d'une note d'information (Tooltip) :

- Le **type** définit le type de données pour chaque entrée de bouton, et la façon d'afficher le bouton. Pour les boutons numériques, cette valeur est une combinaison (**OR** = OU logique) de **INT** (entier) ou de **FLO** (virgule flottante) pour le format du nombre, et de **NUM** (numérique), de **NUMSLI** (numérique à curseur) ou de **TOG** (à bascule 2 états) pour le type du bouton. Les boutons Texte doivent avoir le type **LABEL**.
- Le **nom** est ce qui sera affiché sur le (ou à côté du) bouton. Ceci est limité à 15 caractères.
- L'information d'intervalle est constituée de trois nombres en virgule flottante qui définissent respectivement la valeur par défaut, la valeur minimum et la valeur maximum du bouton. Pour les boutons **TOG**, le minimum est réglé dans l'état appuyé et le maximum est réglé dans l'état relâché.
- La note d'information (ou tooltip) est une chaîne qui sera affichée quand la souris est au-dessus de ce bouton (si l'utilisateur a validé la fonction **Tooltips**). Ceci est limité à 80 caractères, et doit être réglé sur une chaîne **NULL** ("") si elle n'est pas utilisée.

typedef struct Cast {...};
La structure **Cast** est utilisée dans l'appel de la fonction **doit**, et sert de moyen pour accéder simplement aux valeurs des données de chaque plugin. **Cast** doit contenir, dans l'ordre, un entier ou un nombre en virgule flottante pour chaque bouton défini dans **varstr**, en incluant les boutons texte. Typiquement, ceux-ci doivent avoir le même nom que le bouton à référence simple.

float cfra
La valeur **cfra** est réglée par **Blender** à la valeur courante avant chaque passe de rendu. Cette valeur est un nombre
De cellos +/- .5 en fonction du réglage des champs.

plugin_seq_doit prototype
La fonction **plugin_tex_doit** doit être prototypée pour être utilisée par la fonction **getinfo**. Vous n'avez pas à modifier cette ligne.

plugin_seq_getversion
Cette fonction doit exister dans chaque plugin pour qu'il soit correctement chargé. Vous n'avez pas à modifier cette fonction.

plugin_but_changed
Cette fonction est utilisée pour transmettre de l'information sur les boutons qui ont été modifiés par l'utilisateur dans l'interface. La plupart des plugins n'ont pas besoin d'utiliser cette fonction, sauft quand l'interface permet à l'utilisateur de modifier une variable quelconque qui oblige le plugin à recalculer (par exemple, une table de nombres aléatoires).

plugin_init
Si nécessaire, des plugins peuvent utiliser cette fonction pour initialiser des données internes. **Note** : Cette fonction **init** peut être appelée de nombreuses fois si le même plugin de Séquence est copié. N'initialisez pas des données globales spécifiques pour une seule instance d'un plugin dans cette fonction.

plugin_getinfo
Cette fonction est utilisée pour transmettre des informations à **Blender**. Vous n'avez jamais besoin de la modifier.

plugin_seq_doit
La fonction **doit** est responsable de l'application de l'effet du plugin et doit copier les données finales dans le tampon de sortie.

Les Arguments

Cast *cast : La structure **Cast** qui contient les données du plugin, voir ci-dessus l'entrée **Cast**.

float facf0 : La valeur de la Courbe **IPO** du plugin pour le premier décalage (offset) de champ. Si l'utilisateur n'a pas créé de Courbe **IPO**, ceci appartient à l'intervalle entre 0 et 1 pour la durée de l'action du plugin.

float facf1 : La valeur de la Courbe **IPO** du plugin pour le second décalage (offset) de champ. Si l'utilisateur n'a pas créé de Courbe **IPO**, ceci appartient à l'intervalle entre 0 et 1 pour la durée de l'action du plugin.

int x int y : La largeur et la hauteur des tampons (buffers) Images, respectivement.

Imbuf *ibuf1 : Un pointeur vers le premier tampon Image auquel est lié le plugin. Ce sera toujours un tampon Image valide.

Imbuf *ibuf2 : Un pointeur vers le second tampon Image auquel est lié le plugin. Les plugins utilisant ce tampon doivent faire un test pour un tampon **NULL**, car l'utilisateur peux ne pas avoir rattaché le plugin à deux tampons.

Imbuf *out : Le tampon Image pour la sortie du plugin.

Imbuf *use : Un pointeur vers le troisième tampon Image auquel est lié le plugin. Les plugins utilisant ce tampon doivent faire un test pour un tampon **NULL**, car l'utilisateur peux ne pas avoir rattaché le plugin à trois tampons.

ImBuf image structure : La structure **ImBuf** contient toujours des données de pixels en **RGBA** 32 bits. Les structures **ImBuf** sont toujours égales en taille, en fonction des valeurs **x** et **y** transmises.

Interaction avec l'Utilisateur (User Interaction)

Il n'y a aucun moyen pour **Blender** de savoir combien d'entrées sont espérées par un plugin, donc il est possible pour un utilisateur de n'attacher qu'une entrée à un plugin qui en espère deux. Pour cette raison, il est important de toujours vérifier les tampons qu'utilise votre plugin, pour être sûr qu'ils sont tous valides. Les plugins **Sequence** doivent aussi inclure un label texte qui décrit le nombre d'entrées requises dans les boutons de l'interface.

Un Plugin Sequence Générique (Generic Sequence Plugin)

```
#include "plugin.h"
char name[24]= "";

/* Structure pour les boutons,
 * butcode name default min max 0
 */
VarStruct varstr[]= {
        { LABEL, "In: X strips", 0.0, 0.0, 0.0, ""},
};

/* La structure cast sert à l'entrée dans la fonction principale doit
   Varstr etd Cast doivent avoir les mêmes variables dans le même ordre */
typedef struct Cast {
        int dummy; /* à cause du bouton LABEL */
} Cast;

/* cfra : le cellos courant */
float cfra;

void plugin_seq_doit(Cast *, float, float, int, int, ImBuf *, ImBuf *, ImBuf *, ImBuf *);

int plugin_seq_getversion(void) {
        return B_PLUGIN_VERSION;
}

void plugin_but_changed(int but) {
}

void plugin_init() {
}

void plugin_getinfo(PluginInfo *info) {
        info->name= name;
        info->nvars= sizeof(varstr)/sizeof(VarStruct);
        info->cfra= &cfra;
        info->varstr= varstr;
        info->init= plugin_init;
        info->seq_doit= (SeqDoit) plugin_seq_doit;
        info->callback= plugin_but_changed;
}

void plugin_seq_doit(Cast *cast, float facf0, float facf1, int xo, int yo, ImBuf *ibuf1, ImBuf *ibuf2,
ImBuf *outbuf, ImBuf *use) {
        char *in1= (char *)ibuf1->rect;
        char *out=(char *)outbuf->rect;
}
```

Vos Modifications

La première étape consiste à avoir prévu un plan d'action. Quel est le but de ce plugin, que fait-il et comment les utilisateurs pourront-ils l'utiliser. Dans cet exemple, vous allez créer un filtre simple qui aura un curseur pour l'intensité de 0 à 255. Si l'une quelconque des composantes **R**,**G**, ou **B** d'un pixel dans l'image source a une valeur inférieure à l'intensité choisie, il renverra du noir et de l'alpha, sinon il renverra ce qu'il y avait déjà dans l'image d'origine.
Maintenant, vous allez copier notre plugin générique dans le fichier **simpfilt.c**, puis vous allez remplir les parties laissées vides.
C'est toujours une bonne idée d'ajouter quelques commentaires. Le premier indiquera aux utilisateurs ce que fait le plugin, où ils peuvent en obtenir une copie, qui ils doivent contacter en cas de bugs ou de dysfonctionnements, et quelles sont les restrictions de licence sur le code.
N'oubliez pas les /* */ pour encadrer ces commentaires. Les plugins sont écrits en langage **C** et certains compilateurs **C** n'acceptent pas les // pour encadrer les commentaires.

```
/*
Description : Ce plugin est un exemple de plugin de Séquence qui
Filtre les pixels de faible intensité. Il fonctionne avec une Strip en entrée.
Auteur : Kent Mein (mein@cs.umn.edu)
Site Internet : http://www.cs.umn.edu/~mein/blender/plugins
Licence : Domaine Public
Dernière Modification : Samedi 7 septembre 2003 05:57:13
*/
```

Ensuite, vous devez remplir le nom (**name**). Vous devriez conserver le même que celui de votre fichier **.c**. De préférence descriptif, avec moins de 23 caractères, aucun espace, et tout en minuscules.

```
char name[24]= "simpfilt.c";
```

Cast et **varstr** ont besoin d'être mis dans **sync**. Vous voulez un curseur, aussi vous faistes ce qui suit :

```
varStruct varstr[]= {
        { LABEL,     "In: 1 strips", 0.0, 0.0, 0.0, ""},
        { NUM|INT,     "Intensity", 10.0, 0.0, 255.0, "Our threshold value"},
};

typedef struct Cast {
        int dummy;                      /* à cause du bouton LABEL */
        int intensity;
} Cast;
```

Maintenant, vous avez besoin de remplir **plugin_seq_doit**. Vous voulez basiquement faire une boucle sur chaque pixel et si les valeurs **RGB** sont inférieures à l'intensité (**Intensity**), vous voulez que le pixel en sortie soit réglé à 0,0,0,255, sinon il est réglé à la valeur d'entrée pour cette position.

```
int x,y;
for(y=0;y cast->intensity) &&
        (in1[1] > cast->intensity) &&
                (in1[2] > cast->intensity)) {
                        out[0] = out[1] = out[2] = 0;
                        out[3] = 255;
                } else {
                        out[0] = in1[0];
                        out[1] = in1[1];
                        out[2] = in1[2];
                        out[3] = in1[3];
                }
        }
}
```

Vous en avez fini avec **simpfilt.c** .

Compilation

bmake est un utilitaire simple (un script shell) pour vous aider dans la compilation et le développement de plugins, et il peut être trouvé dans le sous-répertoire **plugins** du répertoire d'installation de **Blender**. Il est invoqué par : **bmake (nom_du_plugin.c)** et il essayera de lier les bibliothèques convenables et compilera correctement le fichier C spécifié pour votre système. Si vous essayez de développer des plugins sur une machine **Windows**, **bmake** ne fonctionnera pas et vous devrez utiliser **lcc**. Pour compiler un plugin avec **lcc**, vous pouvez utiliser la méthode suivante après avoir vérifié que vous avez vos plugins dans le répertoire **c:\blender\plugins**. Vous avez ici un exemple vous montrant comment compiler le plugin de Séquence **sweep.c**. Ouvrez une console **Dos** et exécutez ce qui suit :
(**Note** : Vérifiez bien auparavant que le répertoire **lcc\bin** apparaît bien dans votre variable d'environnement **PATH**).

```
cd c:\blender\plugins\sequence\sweep
lcc -Ic:\blender\plugins\include sweep.c
lcclnk -DLL sweep.obj c:\blender\plugins\include\seq.def
implib sweep.dll
```