CDRIN

CENTRE DE DÉVELOPPEMENT
ET DE RECHERCHE
EN IMAGERIE NUMÉRIQUE

TECHNICAL REPORT

E*D Films

Project

Link between Photoshop and Maya

1718_26_EDF

v. 2018-11-15

TABLE OF CONTENTS

INTRODUCTION

This section presents the starting point for the partnership with E*D films for the realization of the Innovation passport.

CONTEXT

E * D Films is an animation film production house and a studio offering full production and postproduction services. Whether in 2D, 3D, VR or stop-motion, they develop intellectual properties, concepts and characters as well as tools and tutorials for the animation and video game industries.

The process of modeling a 3D model in Maya based on an input PSD file is essentially manual. The process is long and requires exporting each layer. Some tools allow a mesh generation from texture, but the result is a triangle mesh. Adapting this result take more time than creating the mesh manually.

Therefore, in the animation industry, where the created mesh must be a quad base mesh for better animation and lighting, there is a need for an efficient quad base mesh from PSD file generation tool.

The first step of the project is a prototype that generates a basic quad mesh and the second step is to create the link between the mesh generation and all the components associated with a mesh. The result must be improved to be used in production. At this step, we have to improve the generation and give a better interface between Photoshop and Maya to save time in production.

OBJECTIFS

- A solution was found to create the connection between Photoshop and Maya.
- Paths are created from the different layers on the export step.
- Nomenclature checked before the export to Maya, with pertinent feedback.
- The mesh generation creates a quad base mesh.
- A user friendly interface gives the control on the generation and correctly feeds all the modifications in the PSD.
- A plugin in Photoshop helps the user to export the path for the Maya tool.
- Additional curves influence the generated meshes.

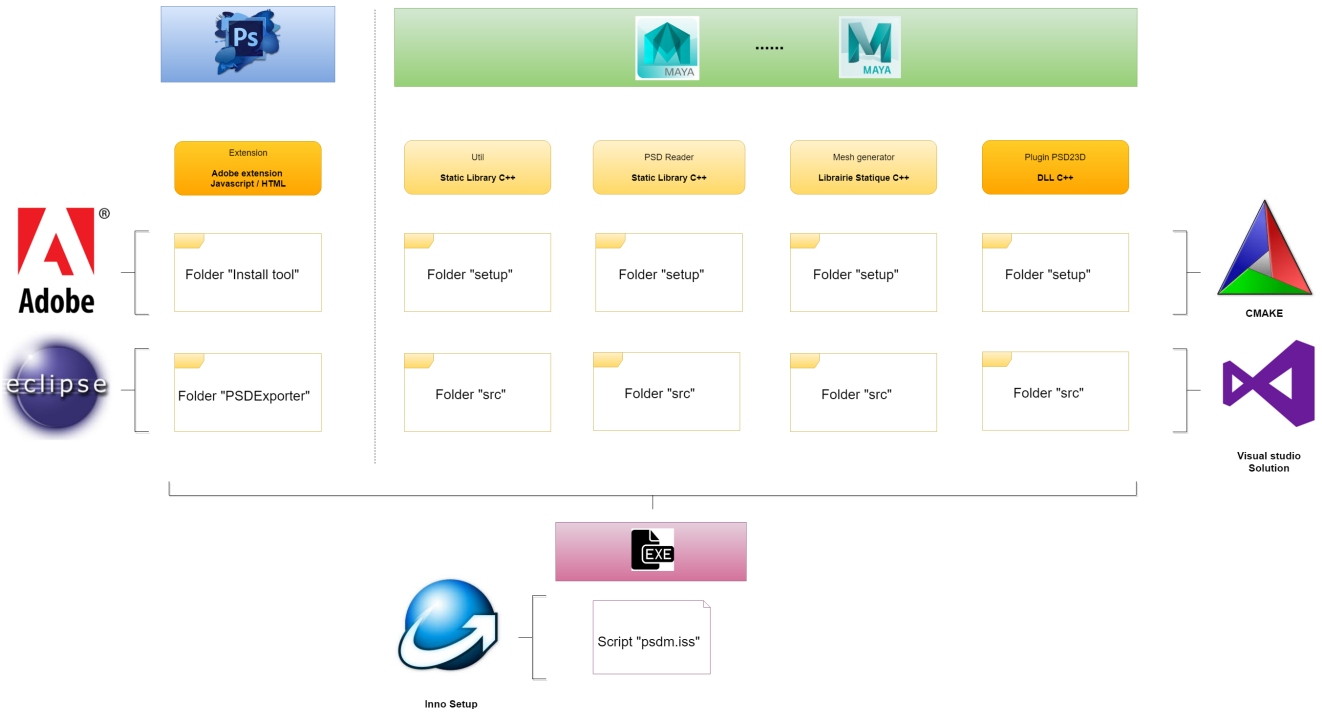
PROJECT STRUCTURE AND DISTRIBUTION

Introduction

The project links 4 differents technologies, our main actors are Photoshop and Maya. The other two technologies are CMAKE for maintaining the project structure and Inno setup for distributing modules to users. The following document presents the architecture of the development environment.

Project diagram

General



The project produces 3 packages at the end:

- A Photoshop extension. (in blue in the graph)
- An "mll" plugin for Maya. (in green in the graph)
- An executable that integrates the two previous packages. (in purple in the graph)

Photoshop

For photoshop we use Eclipe which proposes a plugin for the generation of a template for the adobe extensions.

Maya

For Maya the project is divided into 4 projects:

- **Util**
  - Contains classes shared between all modules and data structures.
- **Psd_reader**
  - Reads the contents of a PSD and stores these values.
- **Mesh_generator**
  - Contains the implementation of several algorithms for generating vertexes and polygons based on different types of input data.
  - Features algorithms for manipulating Bezier curves.
- **Plugin_PSD23D**
  - Contains the implementation of all the management features of a C ++ plugin in maya.
  - The modules for generating the various editor components for creating and manipulating mesh or texture objects in Maya.
  - The implementation of the interface with Qt for using the plugin.
  - The texture conversion algorithm in PNG format.
  - The algorithm for serializing the metadata associated with the parameters of the interface.

Each project has script for use **CMAKE** for independent solution generation of the different maya subprojects.

The **setup** folder makes it possible to generate "visual studio" solutions for each project. These projects allow the generation of independent libraries that can be reused.

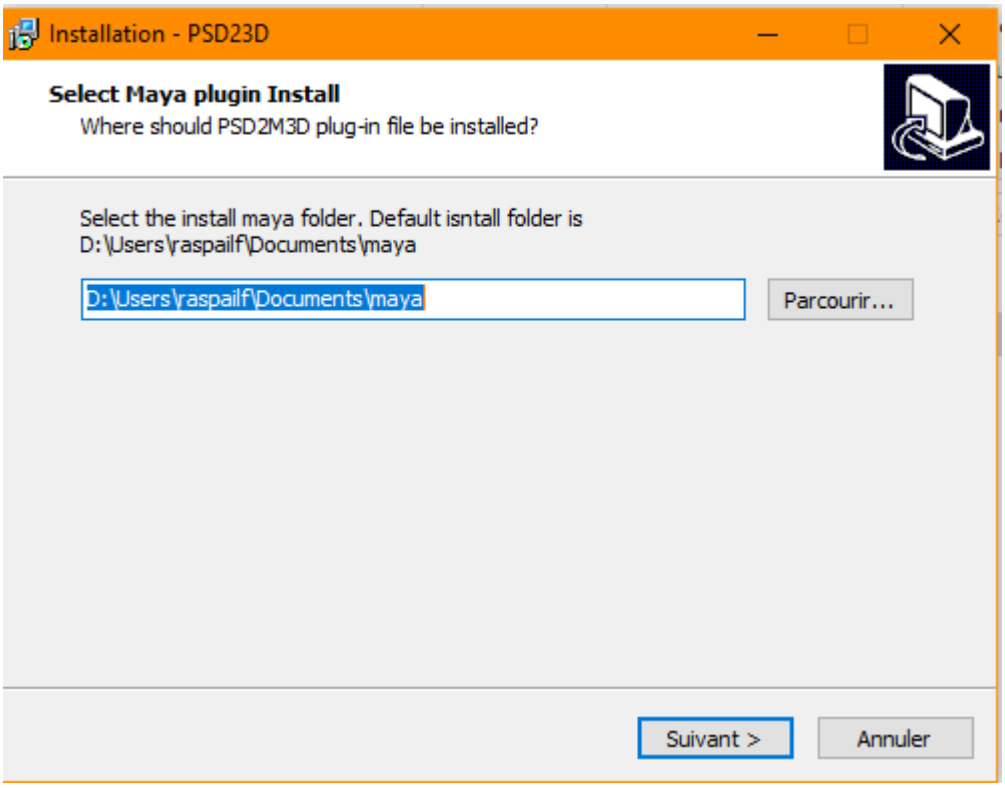
Distribution

For the distribution of the differents packages of the project, we used **Inno setup** allowing to regroup the extension Photoshop as well as the different versions of the plugin Maya (Plugin Maya 2018 and Plugin Maya 2016).

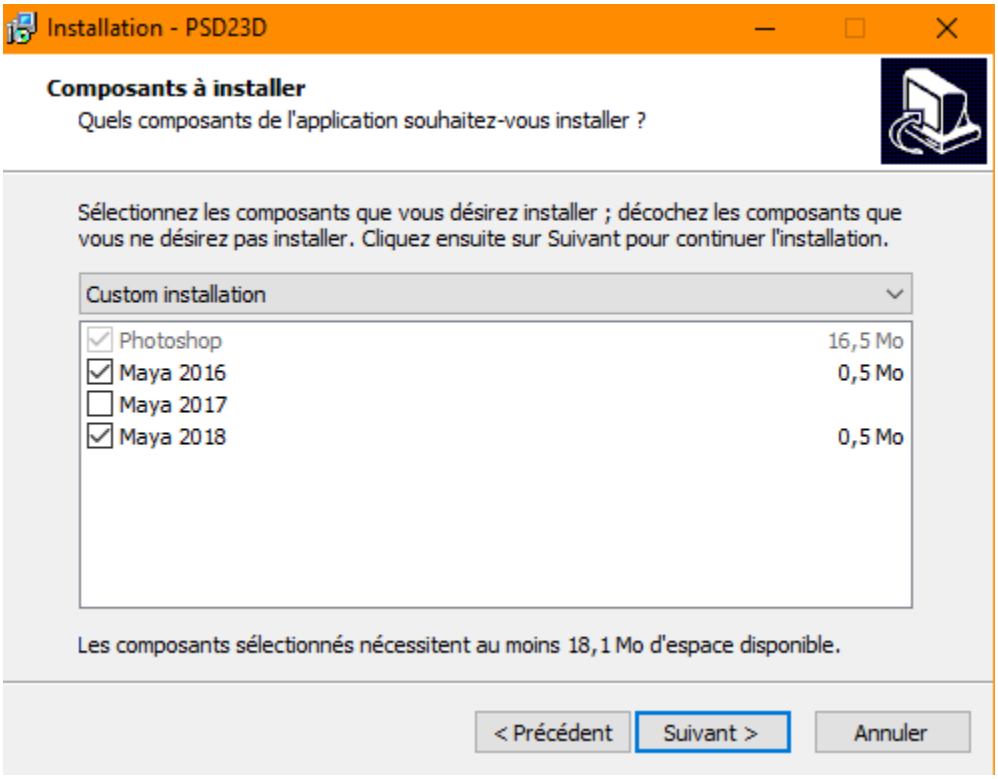
The installer offers deposit options for the plugin and runs a script to install and refer to Photoshop via the extension manager installed with the Adobe Creative Suite.

The executable is generated with a script that offers the ability to:

- Set up custom screens.



- Set the content to integrate during an installation.
- Set the content to install.



- Execute command line operations.

The documentation explains very well the features and scripting API.

- http://www.jrsoftware.org/isinfo.php
- A set of attributes allows you to specify the content and their use [**Setup**], [**Languages**], [**Types**], [**Components**], [**Files**], [**run**], [**Code**].
    - [**Setup**] → Installer global variable, ex: *version*.
    - [**Languages**] → The different languages offered to the user.
    - [**Types**] → The definition of the different types of installations ex: *"custom", "Full"*.
    - [**Components**] → A tag to be used for the file or operation association.
    - [**Files**] → The files to integrate with the installer and thus to deploy during the installation.
    - [**run**] → The command lines to execute for specific operations.
    - [**Code**] →Area to specify the screens, the navigation flow and the interventions of the different operations previously referenced.

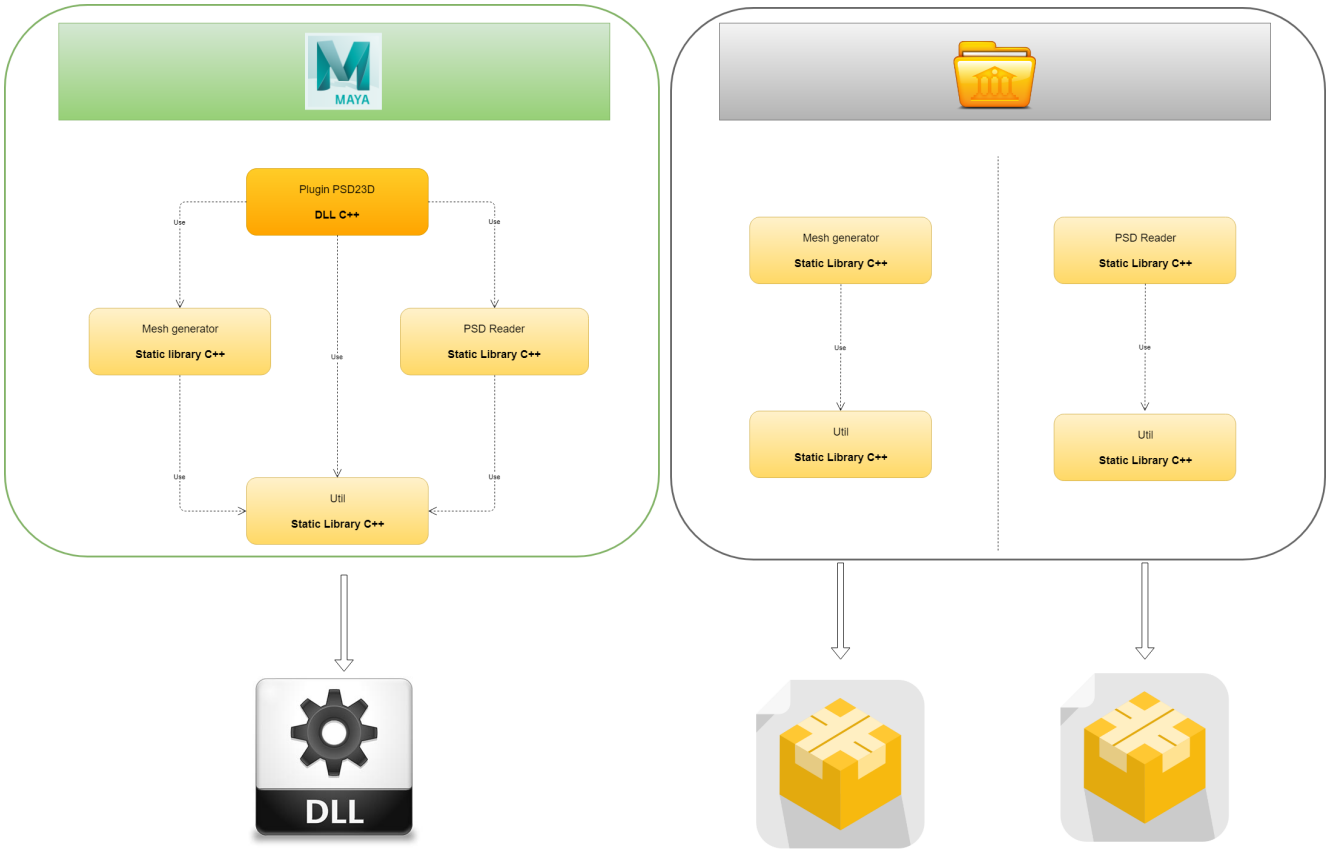
CMAKE AND C++ PROJECT

Introduction

The Maya plugin has been thought for the components project can be used individually. Separating the portion that handles the Maya components from the "psd reader" and "mesh generation" parts allows these modules to be used in other projects. Example: We would be able to write converters from Photoshop to **Unity, Unreal, blender, ...**
Having this type of structure requires a lot of settings for a development environment. So we used a tool to script the setting. CMAKE is the tool selected for this project. It offers the advantage of saving time in changing the structure of the projet for all the team and in prevision of the technology transfer.

Dependency management

Construction plan of the pacqages and their **dependence**.



The fragmentation of our development projects makes it possible to create static libraries that can be used in other projects.

The Maya plugin only uses these libraries.

- The "**PSDparser**" converts and stores information in C ++.
- The "**Mesh generator**" takes curves and input points and returns a vertex and polygon structure exploitable by any tools.

The "**Util**" project groups:

- Mathematical operations.
- Data formats used in reading the PSD file and during generations.
- This project is similar to the notion of "Core" with minimalist classes useful in the different modules.

As we can see in the graph above

- "**Util**" is integrated in all solutions.
- "**Mesh generator** and **PSDparser** " are independent and autonomous modules.
- "**PSD23D**" is the master project with the use of libraries and integration of results in maya. It has a dependency on all previous modules.

Presentation of CMake

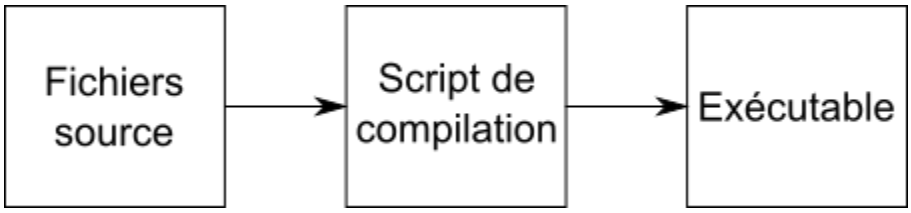
Cmake is a usefull tool for compilation, it allows to:

- To script the **composition** of a project. (script .cpp, .c, .h, organIzed in folder)
- Script the compilation with the management of the different dependencies.
- Centralize parameter variables for compilation.
- Easily create and set up a working environment for Visual Studio or other development environment.
- Manage a solution with multiple interdependent project.

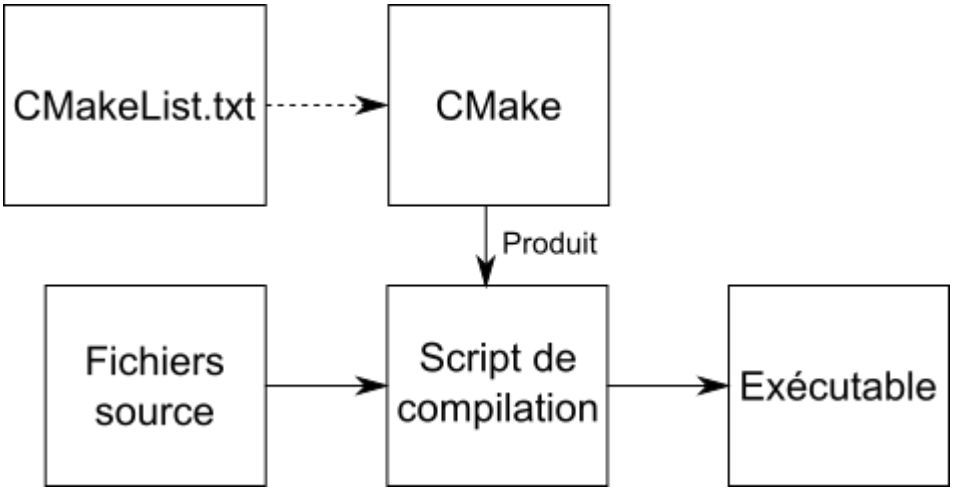
Link to the tool:

- https://cmake.org

Classic Project



Project With CMAKE



Easy management of platforms and versions

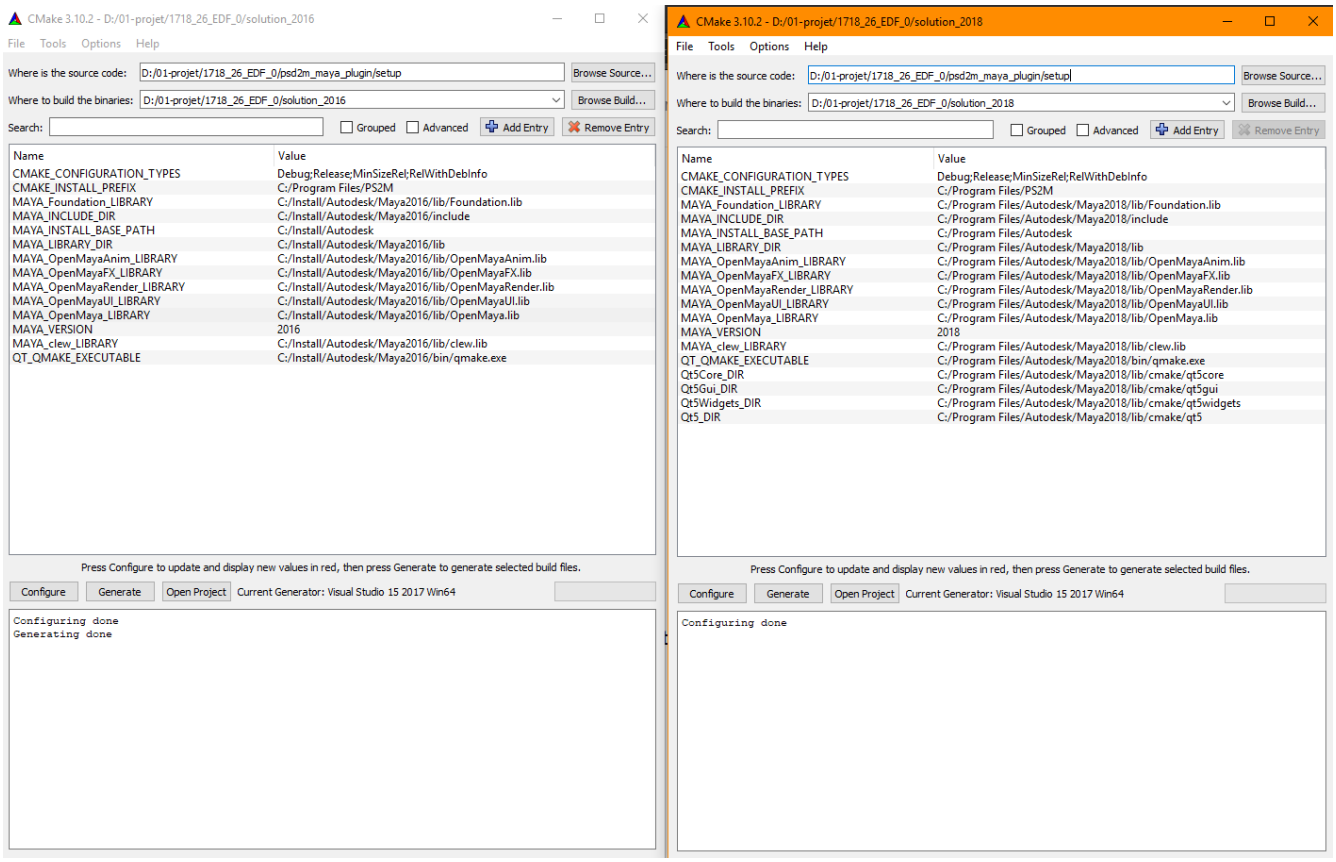
As part of the project we had to deliver the plugin for Maya 2016 and Maya 2018. There are two versions with two different "**devkit**", the **APIs** are different, the **dependencies** are too. A dependency is the use of libraries exposed by Maya or other third-party software.

Moreover Maya is based on **Qt** to make its menu, between the 2016 and 2018 version Maya has gone from version 4.8.6 to version 5.6.1 of Qt. These versions do not follow the standard versions of Qt, they are slightly modified for Maya's needs. In the project no difference between the official documentation of Qt and the components used was noted.

In short our dependencies are:

- the maya API for using Maya components.
- The Qt API embedded in Maya, for the menus.
- Our inter-module dependencies previously presented.

The following illustration shows the generation of a visual studio solution left for "Maya 2016" and right for "Maya 2018".



We observe that:

- The source project is **the same** in both cases.
- The destination folder is different to have the two solutions separately.
- There is need for less parameter for Maya 2016 than Maya 2018.
    - CMake uses the **CMakeLists.txt** scripts to search for dependencies to the libraries used.
    - The exposed variables are the values he automatically found for the dependencies targeted by the CMakeLists.txt scripts.

These variables are useful, as in our case by specifying the "**MAYA_VERSION**" field the dependencies to the maya API and the API of Qt will be different.

If variables are not correctly identified, it has given you the option to specify the value. Useful during unconventional installation of an involved library. In the example above you will notice that my version of Maya 2016 is installed in a folder other than the default ones of Windows. The script will not find a Maya folder to the classic path of an installation, I will have to enter the path to the installation manually then re-click on "**Configure**".

Conclusion:

- A unique project.
- Development environment variables associated with the source code via CMAKE scripts. (no need for environment variable in Windows, less risk of errors)

- Automation of the generation of the solution or the compilation according to the variables contained in the CMakeLists.txt files.

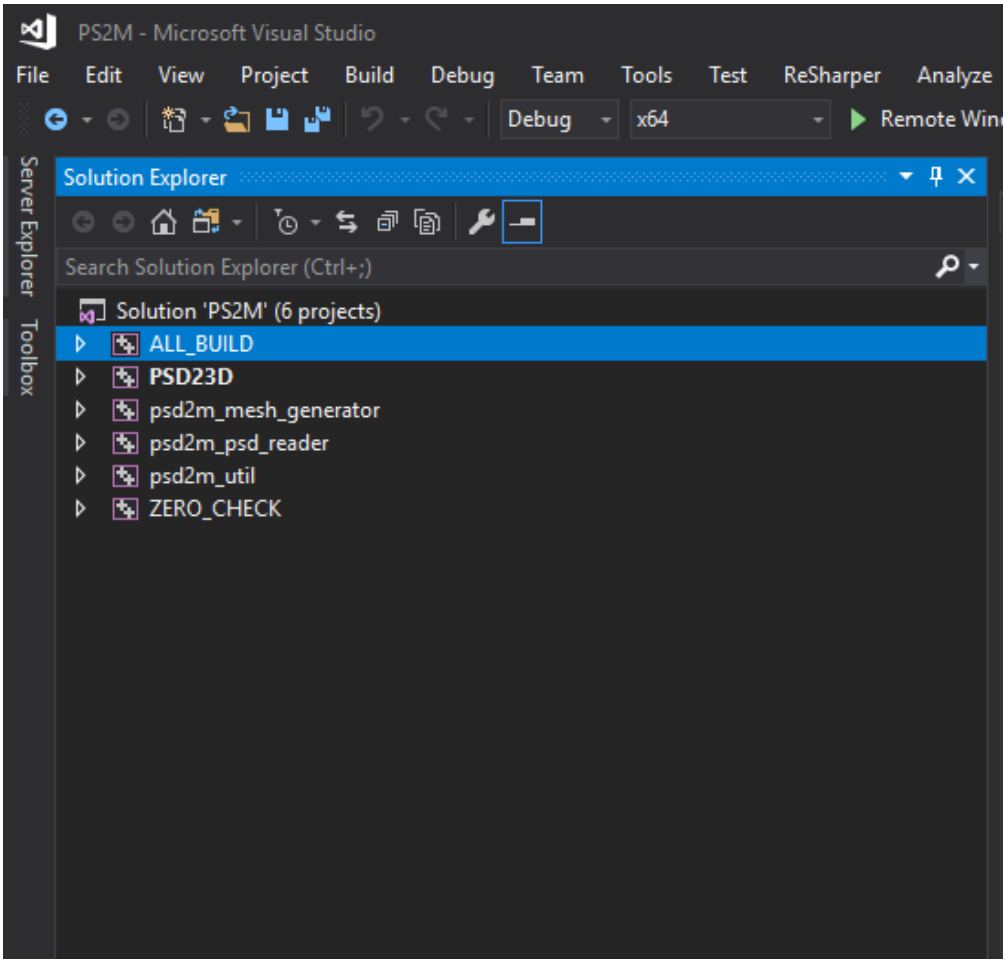
Advantage Independent project generation

Another important point in our project is the presence of a "**Setup**" directory for each project. This folder contains the reference CMakeLists.txt. to configure the project and its dependencies autonomously.

In our case the **setup** of:

- **Util** will only create a solution with Util source code.
- **Mesh generator** and **Psdreader** will respectively have the code of their project as well as that of util being in direct dependence.
- **Psd23d** will have all the projects, being dependent on the three previous ones.

The following image illustrates the solution generated with the "**setup**" folder of **Psd23d**.

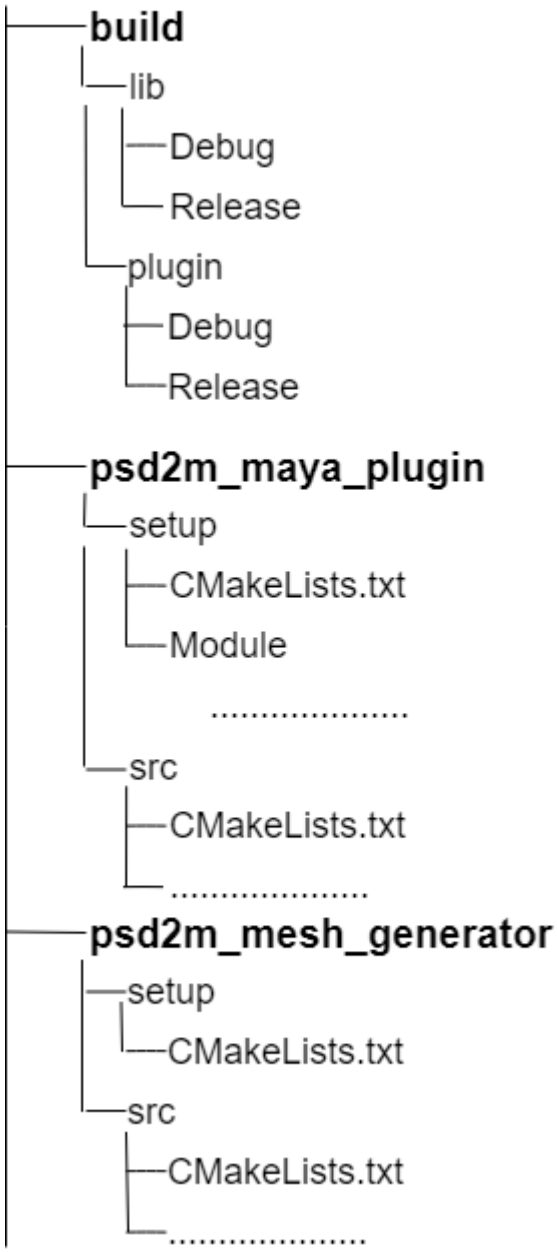


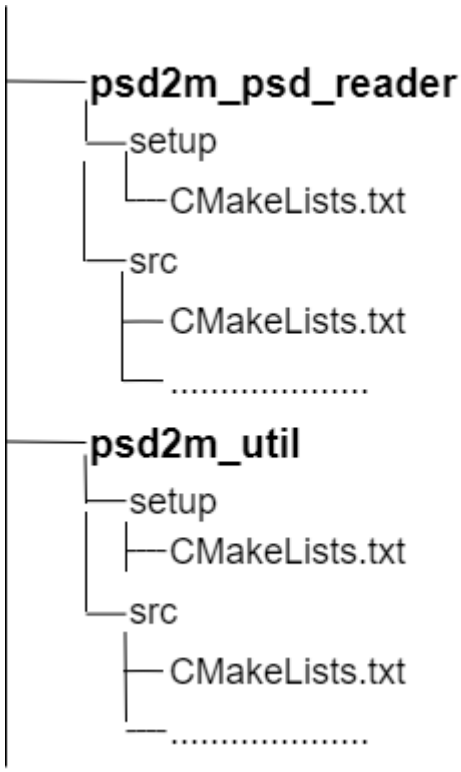
This configuration allows us to easily debug a project and its dependencies.

Zero_check is a project generated by CMAKE, it uses CMakeLists.txt scripts. It re-compiles the environment without going through the CMAKE interface or command lines. Very easy to push a modification to the other programmer who will only have to re-compile **zero_check** to see the modifications of the solution apply. (compilation variable / project structure / adding delete script / ...)

Directory Structure

Here is the organization of directories and projects.





Folders can generate just project solutions and dependencies.

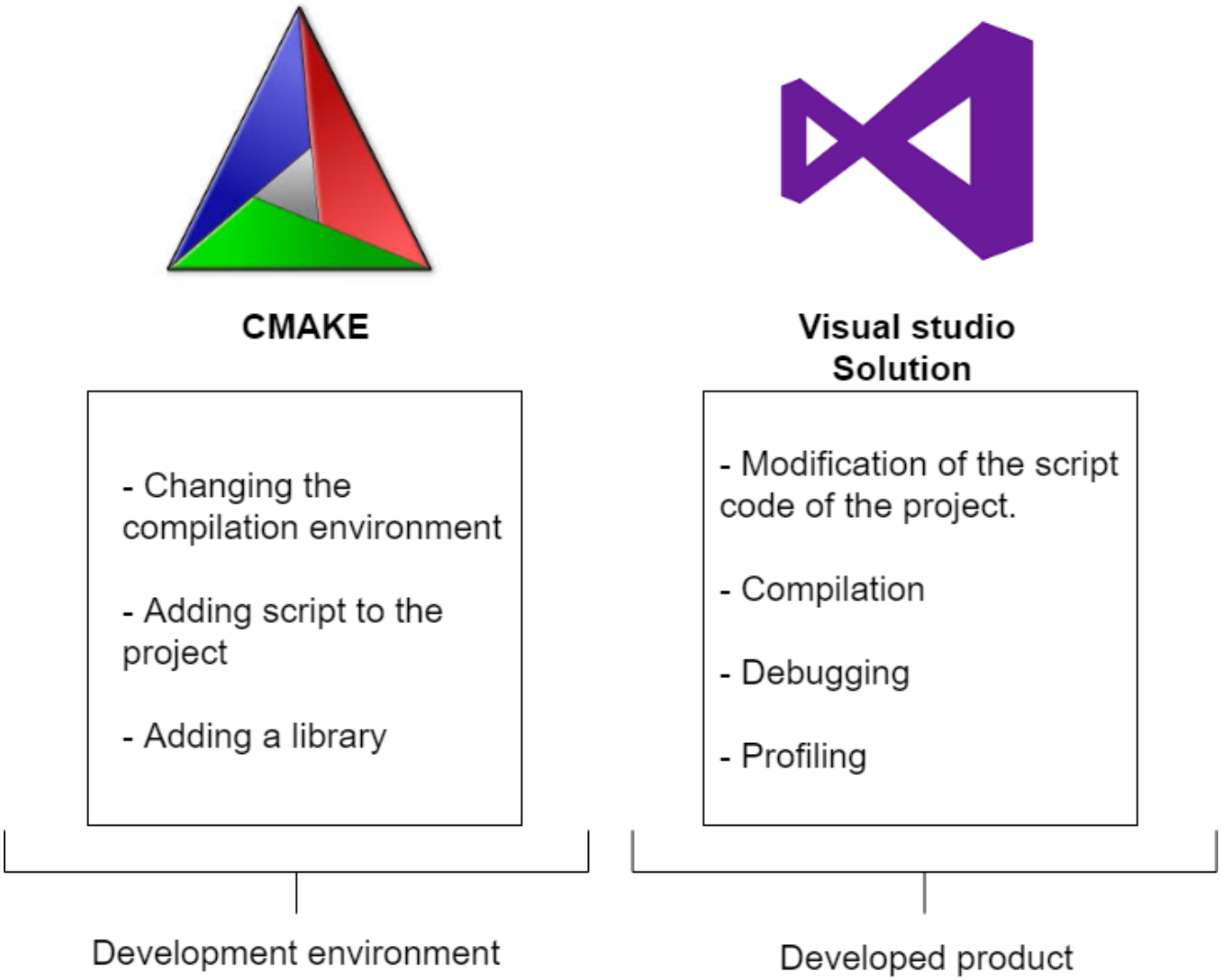
The source folder (**src**) contains the source code for each project

The structure of the files in the directory is different from the one of the solutions, are the CMakeLists.txt which define it.

The **build** folder is a folder specified in CMakeLists.txt.

- This value will be automatically injected into the generated solutions.
- Having the build folder outside projects allows you to automatically exclude them from the versioning system.

La logique de CMAKE



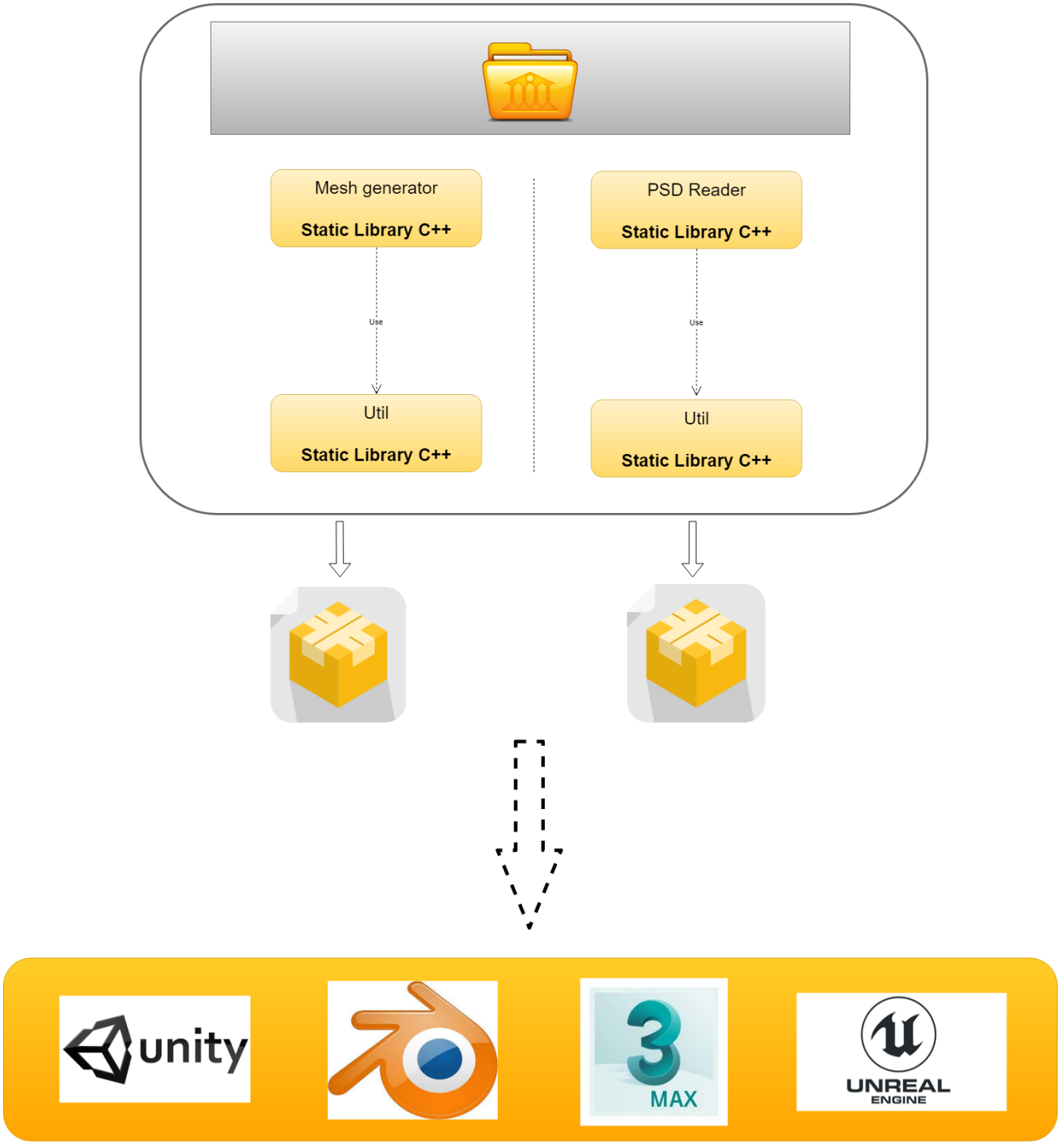
CMAKE is in charge of configuring a project, so all the information implying a change in the structure of a solution must be entered in a CMAKES script file to see them apply in the solution or at the compilation:

- Adding a new library. (dependency management)
- Add a script.
- The compilation options.

Then our development tool will only serve:

- Change the code of the files.
- Compile.
- Debug / Profile.

The possibilities

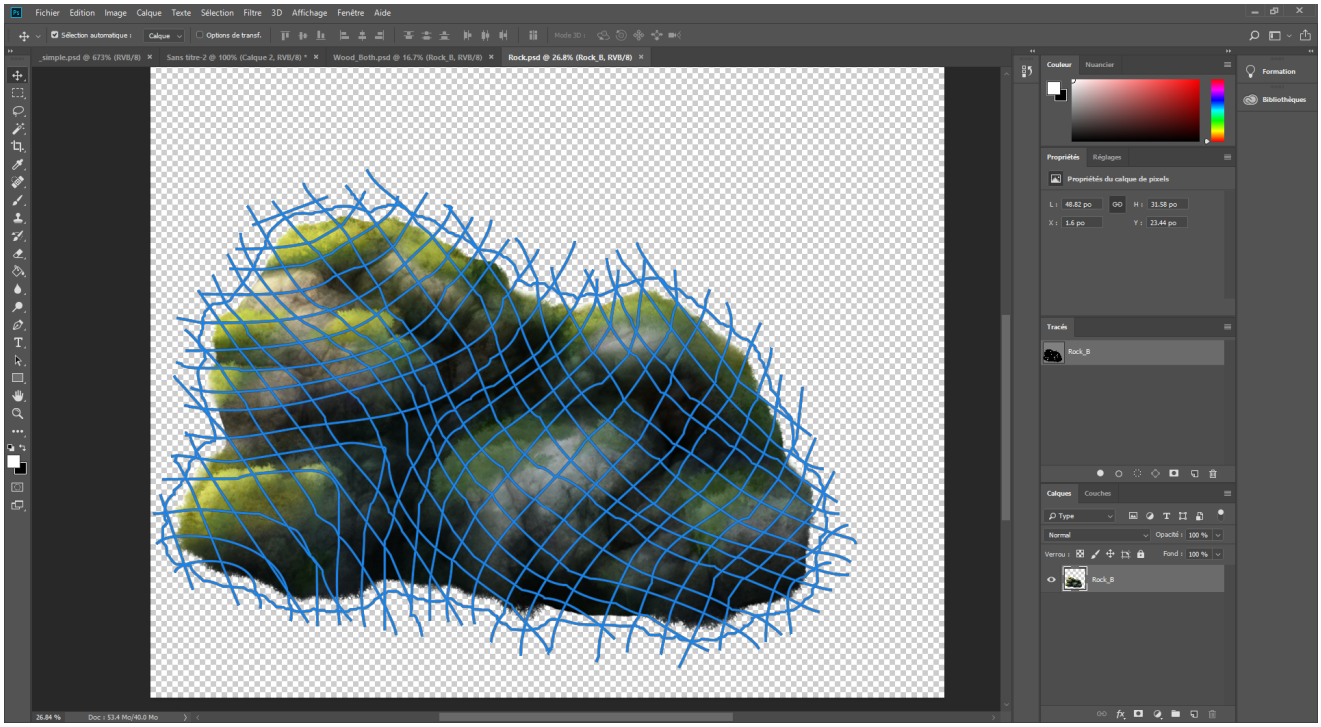


The concept of reading PSD data remains unchanged from platform to another and when generating data from a mesh. Just write the equivalent of **PSD23D** for each environment presented above.

CURVE ALGORITHM

Introduction

The Curve Algorithm allows a user to trace curves in Photoshop and use these created grid to create a mesh and its faces.
Here is an example of a possible grid :



This algorithm is split in 4 large steps.

1. Find all intersections between each curve.
2. Find all the neighbours of each intersection.
3. Clean up the intersections.
4. Find all the faces of the mesh.

After these 4 steps we'll have more than enough information to create the final mesh that will be sent to Maya.

These steps and all the code related to the algorithm can be found at it's entry point in the GenerateMesh method in the script curveMeshGenerator.cpp. It is recommended to follow this documentation with the script beside.

The Intersections

Before we start we should define a few classes :

- **Vector2F** : A position in space defined by two floating point values, X and Y.
- **Bezier** : Four **Vector2F** that define a Bezier Curve.
- **Curve** : The combination of one or many **Bezier** following each other. (Ex. the last point of Bezier 001 will be the first point of Bezier 002, etc.)

We start by creating the data structure "*std::vector<CurveData> paths*". Which leads us to define more classes :

- **CurveData** : Contains a **Curve** and a *vector* of **PathIntersection**.
- **PathIntersection** : Contains a value that represents its position in a **Curve** and the **Node** that represents the intersection.
- **Node** : Contains the **Vector2F** of the intersection, the index of the node in its data structure and all its neighbours.

Once we have create all our *paths*, we will iterate through each **CurveData** to find and add the intersections. Each **CurveData** needs to be compared with every other **CurveData** (including itself) to make sure we find all intersections. But we do not need to compare a path twice (ie. A with B and then B with A will have the same results).

As we can see in the method **FindIntersections**, each intersection found will create three new objects. Two **PathIntersection** that will be added to each **CurveData**. And a single **Node** that each **PathIntersection** references. Each **Node** is added to the data structure "*std::vector<Node*> & nodes*".

Next, we will slightly study the algorithm used to find intersections between two bezier curves. This part of the code in the project has been taken from https://github.com/erich666/GraphicsGems/tree/master/gemsiv/curve_isect and the creator of this code based himself off https://www.particleincell.com/2013/cubic-line-intersection/. Which gives us the following pseudo code:

```
class Bezier:
        p0, p1, p2, p3

func BezierIntersection(Bezier A, Bezier B):
        // Depth is found based on Wang's Theorem (https://vdocuments.site/documents/
flatness-criteria-for-subdivision-of-rational-bezier-curves-and-surfaces.html)
        depthA = A.depth
        depthB = B.depth

        if !BoundsOverlap(A, B):
                return

        RecursiveIntersection(A, B)

func RecursiveIntersection(Bezier A, int depthA, Bezier B, int depthB):
        if depthA == 0 && depthB == 0:        // We can now compare the bezier curves as
if they were straight lines
                return SegmentIntersection(A.p0, A.p3, B.p0, B.p3)

        bezierASplits = depthA <= 0 ? { A } : A.split()
        bezierBSplits = depthB <= 0 ? { B } : B.split()

        --depthA
        --depthB

        forall subA in bezierASplits:
                forall subB in bezierBSplits:
                        if BoundsOverlap(subA, subB):
                                return RecursiveIntersection(subA, depthA, subB, depthB)
```

The neighbours

For all the next steps we will need to know the neighbours of each **Node** / intersections. This step is done quickly as have done a little work in this direction in the previous step.

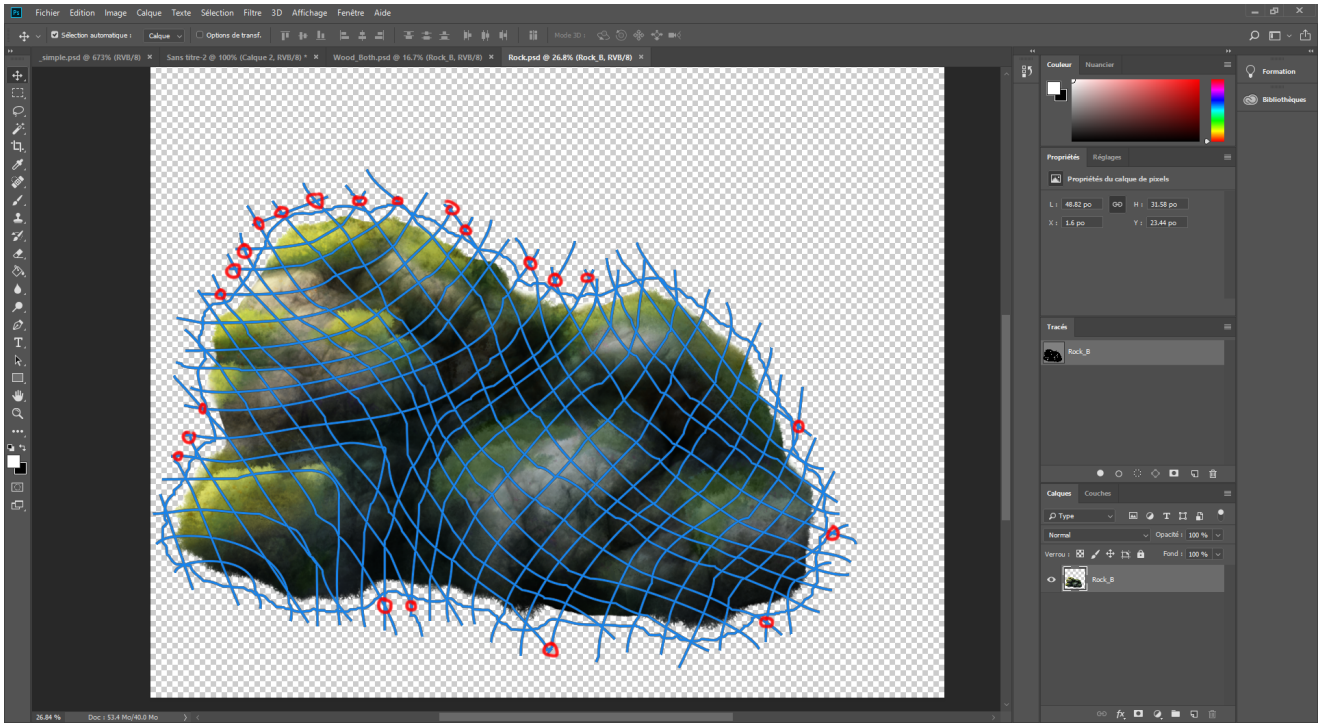
Each list of **PathIntersection** in a **CurveData** if ordered from smallest to largest position in the curve. Next, we connect each **PathIntersection.Node** to the next. We no longer need the PathIntersections in the algorithm as all **Nodes** are now connected.

Clean up

We will now keep care of three problems before continuing to the main algorithm :

Problematique 1

The first clean up is removing all **Nodes** outside the original curve (as we can see in the picture below). Sadly, there is no information that can tell us which curve is the "original" one. So instead we remove all **Nodes** outside all closed paths. If there are no closed paths, all Nodes are kept.

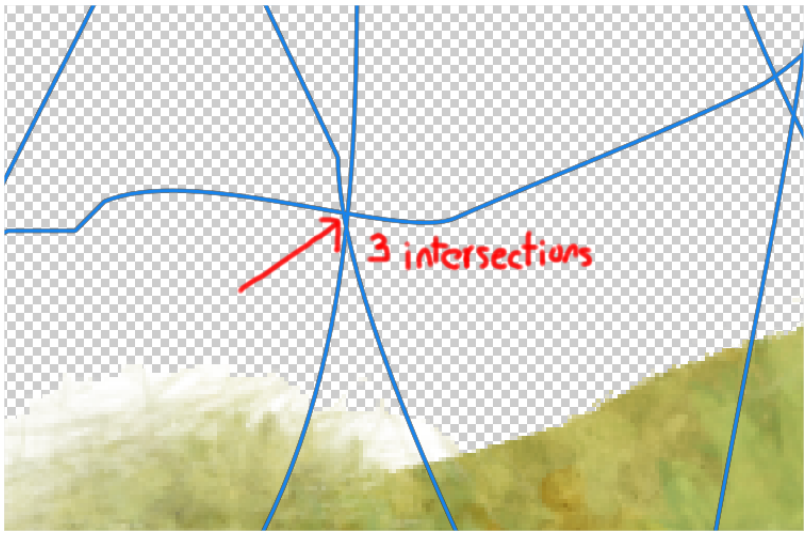


Problematique 2

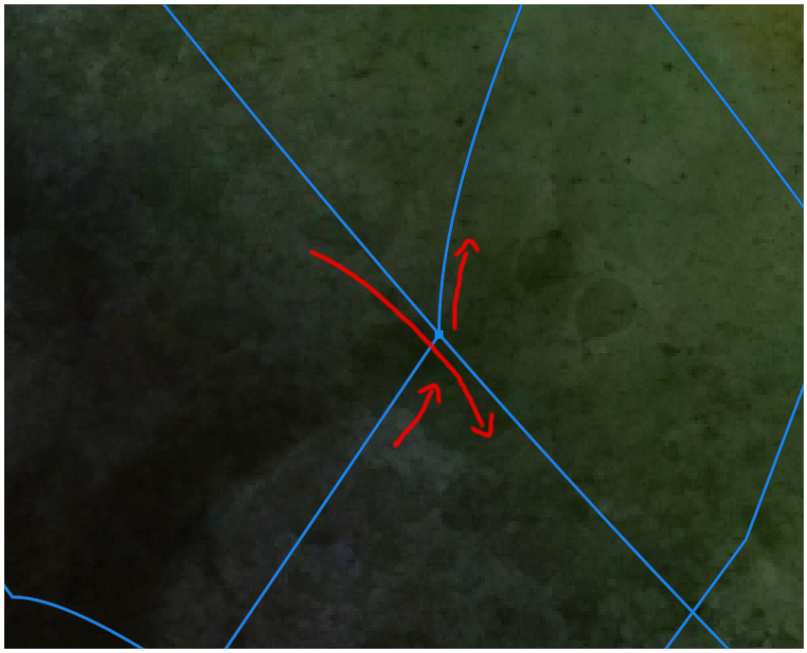
Next, we will combine all intersections that are too close to each other.

There are two reasons for this;

1. Intersections can find themselves really close to each other. This is caused by the grid and it is to be expected as we allow the users of the tool to draw any shape they want.



1. It's possible to have two intersections at exactly the same point. This happens when two bezier curves (let's say B1 and B2) meet each other exactly on a third one (B3). An intersection is found between B1 and B3 and an intersection is found between B2 and B3 so we then have two intersections.



Problematique 3

Lastly, after we have removed all these **Nodes**, we need to reset the index values of each **Node** as they no longer match with the *nodes* structure. So we iterate through all the **Nodes** and reset their indexes.

The Faces

Now we must power through all the **Nodes** to find all the faces. In our case, a Face is the equivalent of a *Minimal Cycle*, which in other words means "a cycle which does not contain any other cycle". The problem of finding all Minimal Cycles is called the Minimal Cycle Basis.

Fortunately, somebody has already solved the Minimal Cycle Basis problem:

- https://www.geometrictools.com/Documentation/MinimalCycleBasis.pdf.

We have implemented this algorithm in the method MinimalCycleSearch.

Here's how it looks explained step by step :

1. Choose a **Node** that is ensured to be outside any cycle. (So we'll choose the **Node** that is most to the left).
2. Choose the **Neighbour** of our **Node** that is the right-most (or the most clockwise) by basing ourselves on a point that is further than the chosen **Node** (as we chose the **Node** completely to the left we can choose a point a little more to the left).
3. Next we choose a neighbour to **Neighbour** that is left-most and we keep looking for left-most neighbours untile we find our starting **Node**.
    a. If we can't find the starting **Node**, it means that we found ourselves in a dead-end. So we can simply abandon the **Neighbour**.
    b. It is possible to find the starting **Node** but to still have a face that isn't valid. Ex. A **Node** is found more than once. In this case we can also abandon the face and the **Neighbour**.
4. After we find a valid face we remove the connection between the starting **Node** and the **Neighbour**.
5. We now return to the second step until the starting **Node** only has a single neighbour. At that point we return to the first step and find a new starting **Node**.

Here's how it looks in pseudo code :

```
func MinimalCycleSearch:
        faces = {}
        nodes.SortBy(x position)
        forall current in nodes:
                if current.neighbours.empty:
                        continue

                leftPos = current.pos - (x:1, y:0)
                // While we have more than one neighbour.
                while current.neighbours.size > 1:
                        neighbour = current.GetMostClockwise(leftPos)        // We're
looking here for the neighbour that is right-most of our current
```

```
                        face = {}
                        // Now that we have a first neighbour, we'll start looking left-
most with CompleteFace.
                        CompleteFace(current, neighbour, face)
                        if !face.valid:        // Is the face valid, check out the
method ValidateFace in the script curveMeshGenerator.cpp for more info
                                current.removeNeighbour(neighbour)
                                neighbour.removeNeighbour(current)
                                continue

                        current.removeNeighbour(neighbour)
                        neighbour.removeNeighbour(current)
                        faces.add(face)

                current.ClearNeighbours()

func CompleteFace(first, second, face):
        face.add(first.index)

        previous = first
        current = second
        while current != first:
                face.add(current.index)
                next = current.GetMostCounterClockwise(previous)

                previous = current
                current = next
```

LINEAR GENERATION ALGORITHM

Introduction

One of the first algorithm of the project allowed to quickly generate a mesh by recovering the contours of the object in a layer of "Path" in photoshop.
The algorithm is very practical for some object of linear form (regular object, building, tree, ...).
We kept the algorithm despite the arrival of the algorithm based on the curves of the vector masks.

How the algorithm works

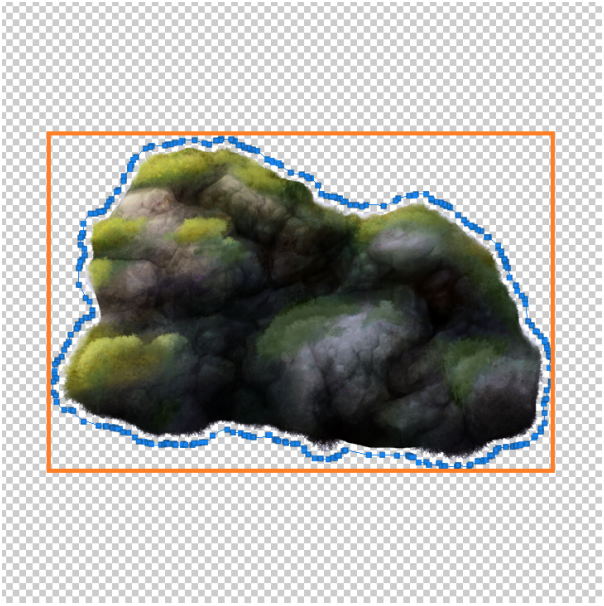
The algorithm works in several steps, it takes to parameter several different values

- The "**Bounding box**".
- The number of polygons desired on the **height**.
- A list of **Bezier curves**.

On retrouve le point d'entrée dans le script **linearMesh.ccp** avec la fonction *GenerateMesh..*

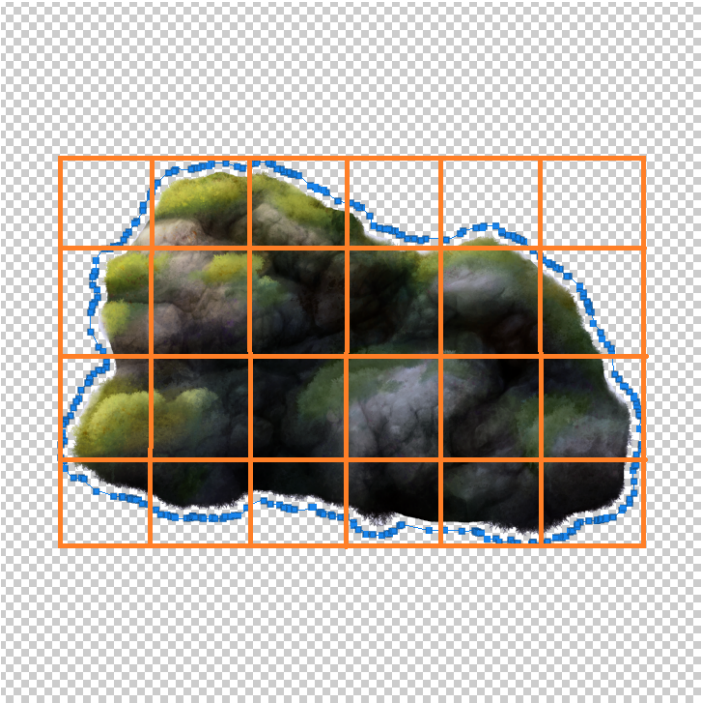
Mesh Precision Parameter

We get the height of the bounding box. We divide this height by the desired number of polygons on the height (value in parameter). We find with this method the size of the edges of our polygons that we will be called "precision".



Generation Of The Grid

We use the point at the top left of the bounding box, then we set points horizontally and vertically at regular intervals of the size of the value calculated in the previous step.



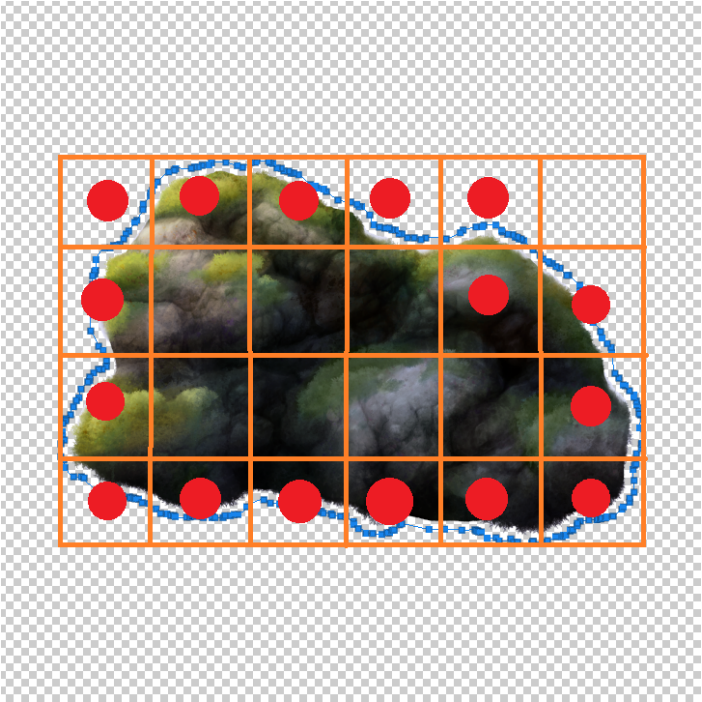
Here is the first step of our mesh generation, we will add all the points in a table representing the vextex. The array is a two-input array, so a polygon at this stage is defined only by the row / column index of the array.

Filtration Of Contours

We will now go through the mesh outline (in blue) and detect all intersections with the grid. That is when a point changes polygon area. It is mainly vectorial calculation to know if a point is on one side or the other side of a vector. The vector being the edge of the polygon at the moment a point is traversed. We take care of the orientation of the displacement to do the calculation, which allows us to do the calculation with the right edge of the current polygon. When a polygon change is detected, we keep the position of the intersection as well as the intersected edge (left, right, up down).

Thus at the end of this step each polygon intersected by the contour curve will contain the information of intersected edges and the value or values of the intersection position.

Now, let's just go through the list of polygons defined by the position of the grid points. So we have the contour polygon list. corresponding to the red dot.

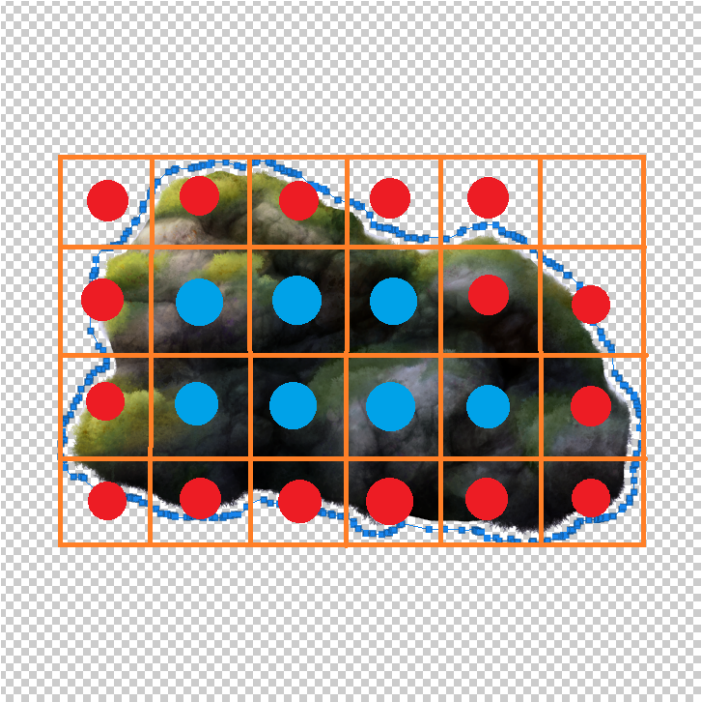


Content Identification

The last step is to identify the polygons belonging to the mesh. To do it we scan each line of the grid from left to right in search of intersection.

On a line if an intersection is detected we identify the following polygons as belonging to the mesh, until we meet the next contour polygon on the same line. The algorithm will then stop identifying the following polygons as belonging to the mesh.

The following polygons will be in the absence of detected intersection, considered as not belonging to the mesh. In case of a new intersection detection on the same line, the previous process will start again.



This process offers the possibility to treat cases of hollow form or with several distinct pixel areas separated by an alpha value equal to 0, on the same layer.

Polygon Table Construction

At the end of this filtration step it is sufficient to browse the identified polygons and fill a MeshPoly structure, containing the indexes of the 4 vertex associated with the identified polygons.

the curves

Depending on the list of points read when reading the PSD we use the following mathematical formula to calculate a series of points at regular intervals.

$$\mathbf{P}(t) = \mathbf{P}_0(1-t)^3 + 3\mathbf{P}_1 t(1-t)^2 + 3\mathbf{P}_2 t^2(1-t) + \mathbf{P}_3 t^3$$

This series of points is useful for the contour path as well as for calculating intersections with the grid.

Bounding box

Calculation Of The Classic Bounding Box

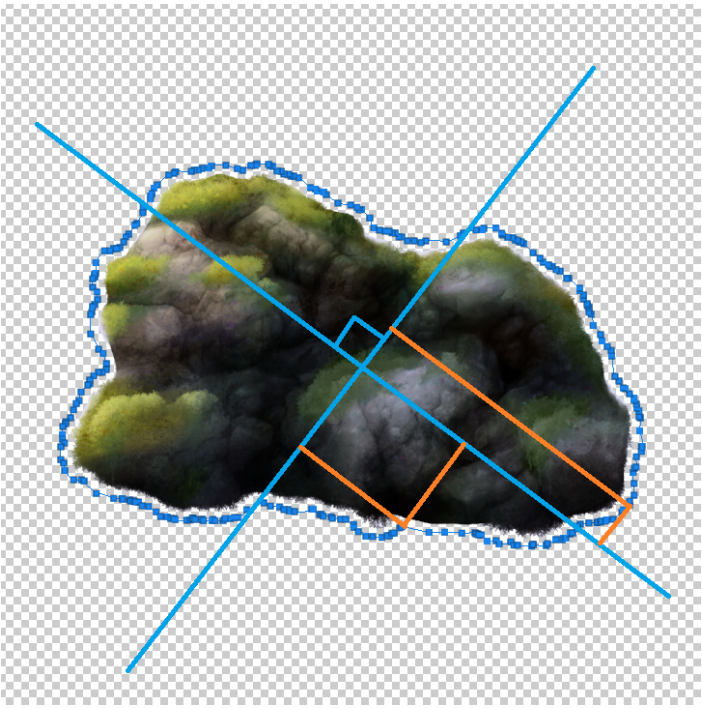
The bounding box is calculated by default according to the horizontal and vertical orientation of the layer. Its obtaining consists in traversing all the points of the curve and identifying the points having the extreme values in X and Y. Once obtained it is enough to take the values and the combined ones to obtain the coordinates of the points making it possible to trace a frame including all the points of the curve.

The notion of **extreme value** means the smallest value and the largest value.

Calculation Of The Bounding Box Oriented

It is possible to give an orientation to the mesh. To do this a parameter gives us the value of an angle. We calculate a vector from this angle and its orthogonal vector.

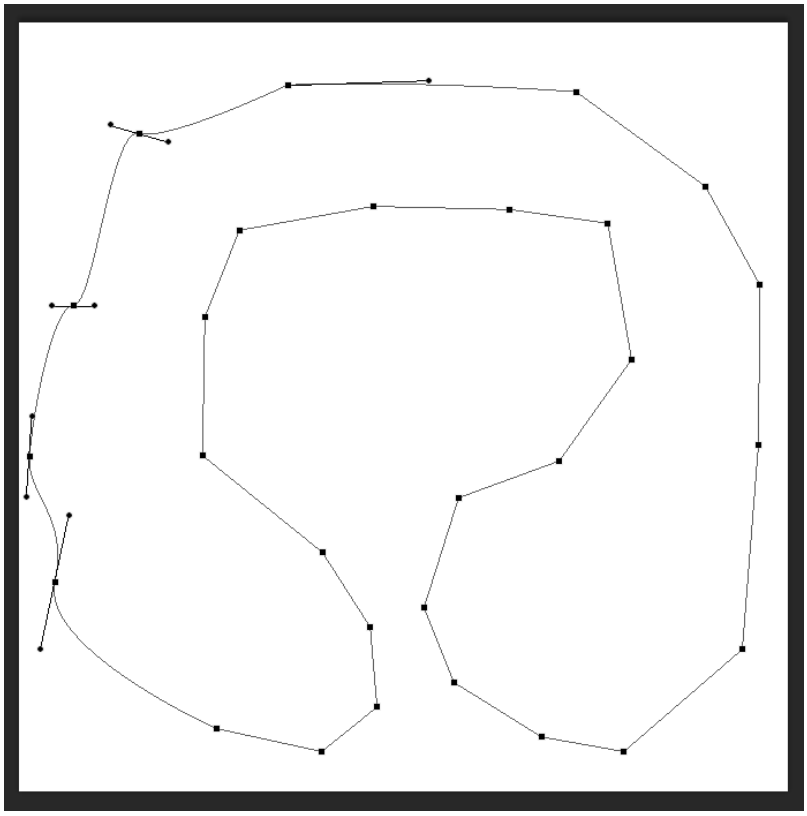
Subsequently we make a projection of the points of the curve on these two vectors. We will then proceed as with a conventional bounding box, we will combine the values of the positions of the projected points at the extremes of each of the vectors. We will thus obtain the points defining the oriented bounding box.

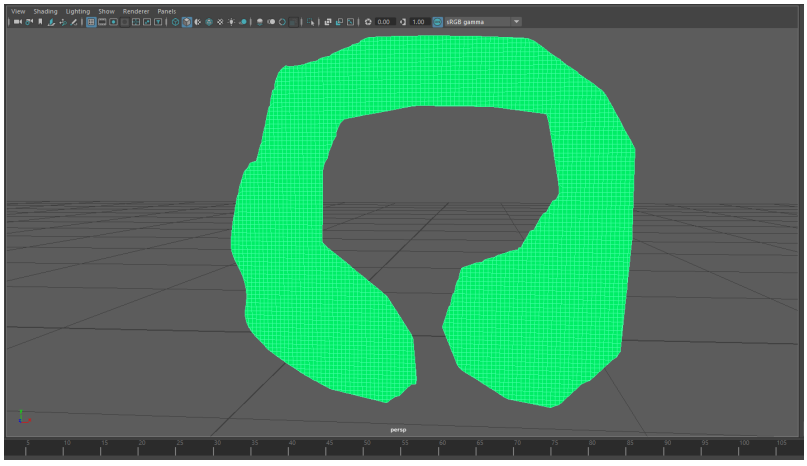


Special cases

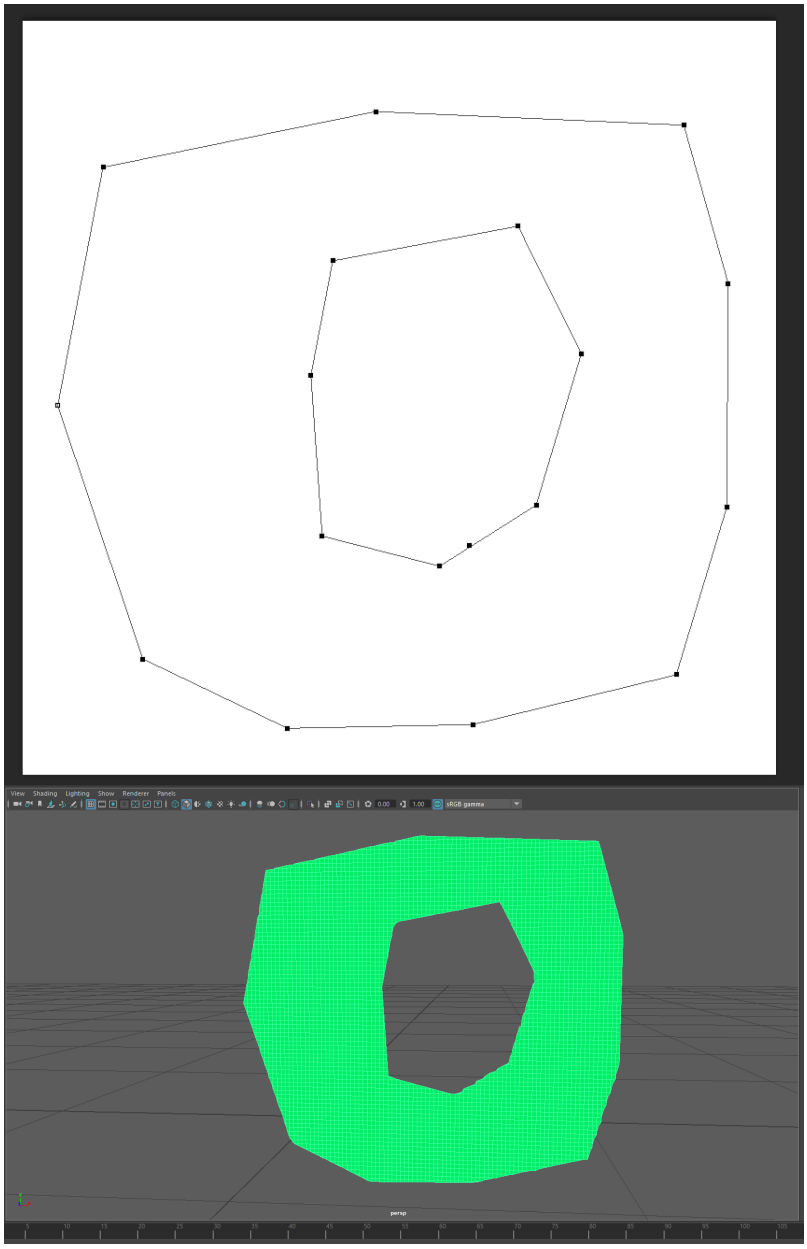
As previously mentioned, the algorithm deals with the following cases

The Hollow Forms

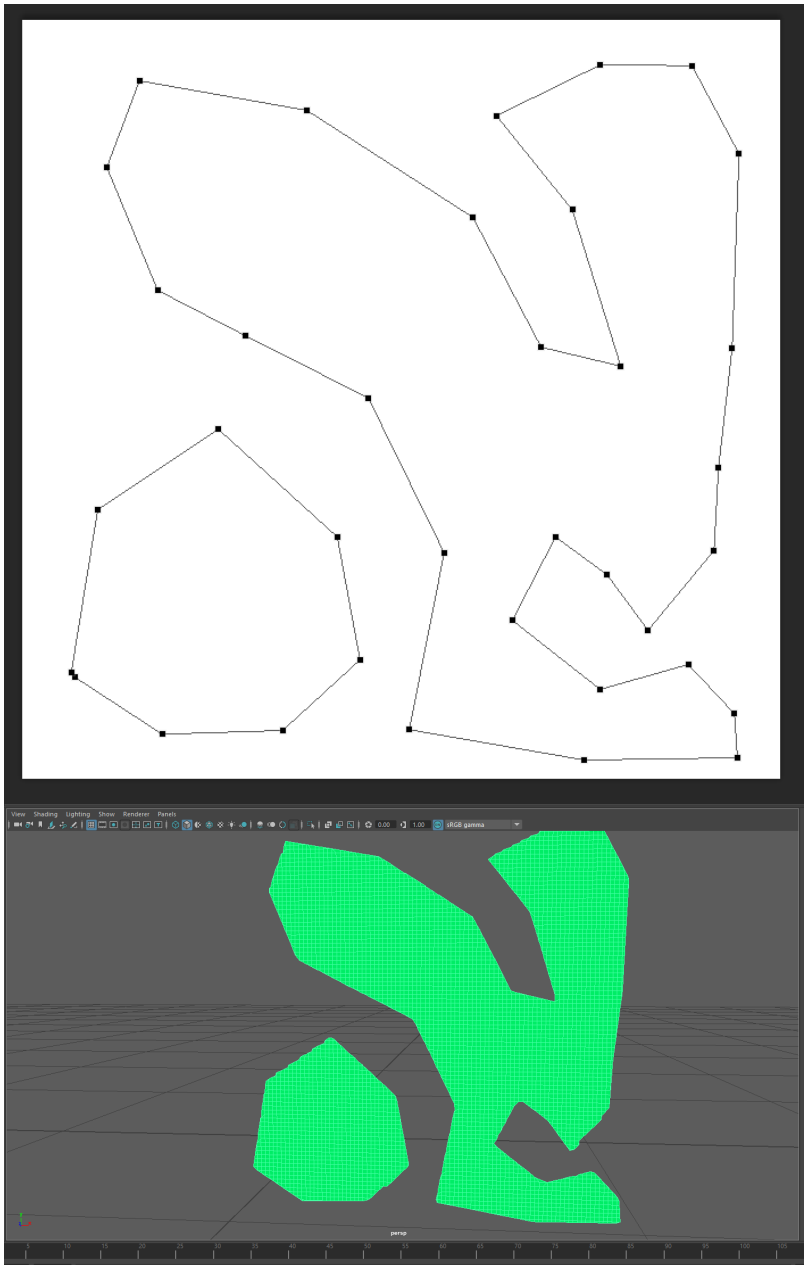




Nested Hollow Shapes



Multiple Shapes By Layer

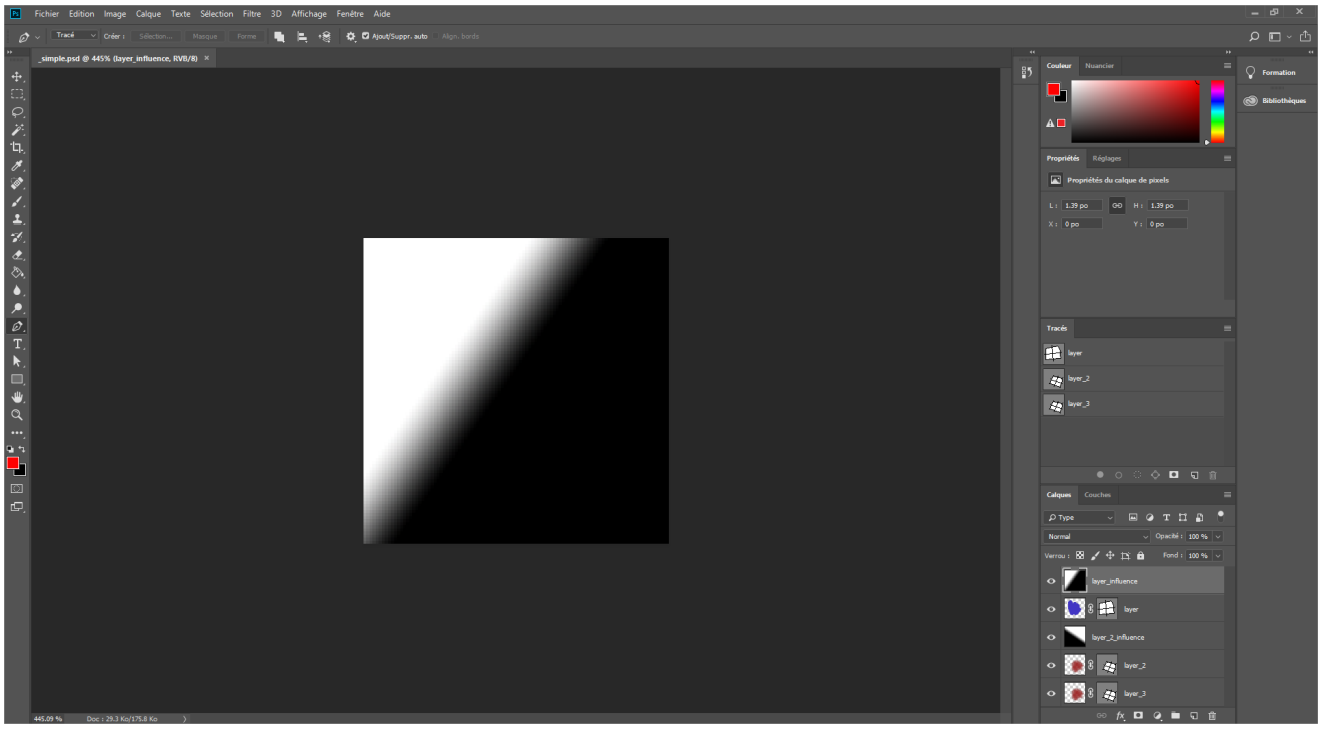


Having the points of intersection it is possible to redefine the position of the points of the polygons to obtain an outline respecting the shape of the curve.

INFLUENCE ALGORITHM

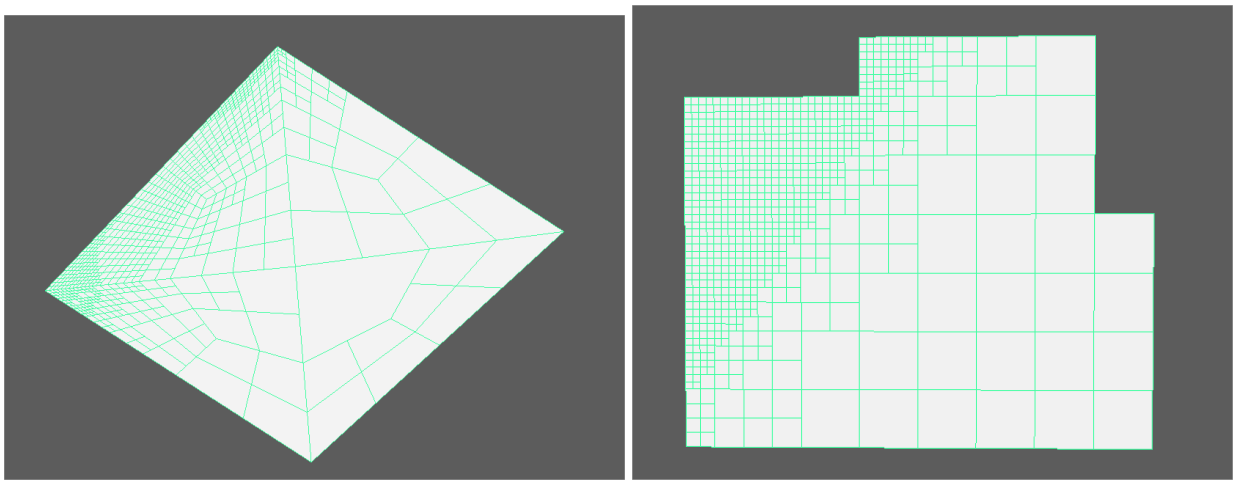
Introduction

Once a mesh has been generated by either the Linear Algorithm or the Curve Algorithm, it is possible to subdivide the faces with the "**Influence Algorithm**". This algorithm uses an influence layer to affect the mesh. This Influence Layer takes the shape of a second layer named exactly like the other layer but with "**_influence**" at the end and all pixels in the layer must be between black and white.
In the following miage, we can see that "layer" and "layer_2" both have influence layers and that "layer_3" does not have any.



The influence layer works like a mask, a face that finds itself under a white pixel will be subdivided (keep reading for more information on how this works).

Here is an example of the effect of this algorithm affected to the layer "layer" (the image to the left has been generated with the **Curve Algorithm**, and the one to the right with the **Linear Algorithm**) :



This algorithm has a few steps which will then be combined to form the complete algorithm.

1. Build a data structure to hold all the faces.
2. Find if a face needs to be subdivied or not.
3. Find the point in the center of a face.
4. Subdivide the face.
5. Warn other faces when edges are cut in half.
6. Combine all previous steps together.

These steps and all the code related to the algorithm can be found at it's entry point in the SubdivideFaces method in the script influenceMesh.cpp. It is recommended to follow this documentation with the script beside.

Building the faces

Before starting our process, we will build our data structure **"std::queue<MeshFace> toProcess**". We do this to increase the readability and decrease the quantity of operations done to the data structure.

- The class **MeshFace** contains the list of indexes (which links to a 2d vertex) and some methods to help the algorithm.

For each **MeshFace**, we call SetShouldSubdivide which will decide if this face should be subdivided or not (see the next steps for more information about the subdivision).

Should we split?

As described in the introduction, an influence layer is used to divide a face. Here is a little pseudo code that explains how the values are used :

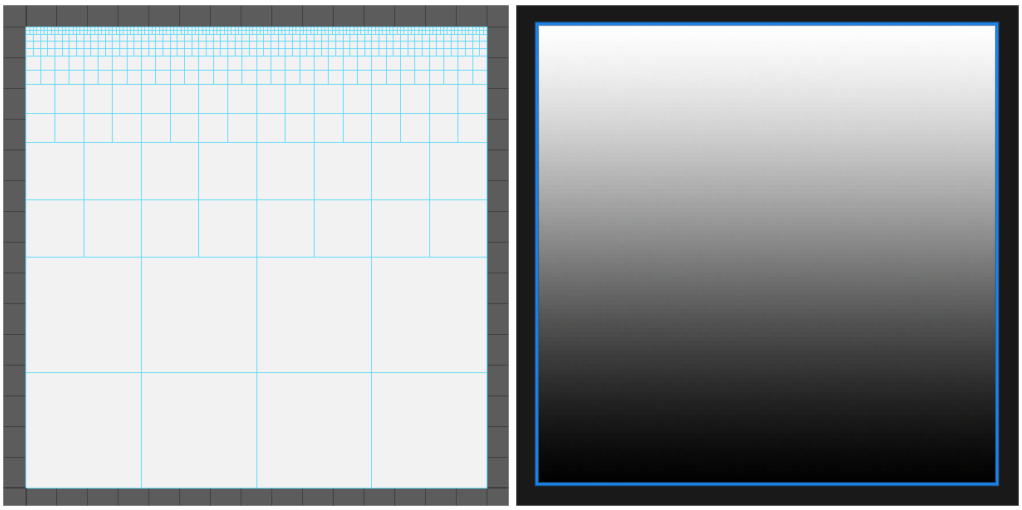
```
// A 2D box in which the face is included
boundingBox = GetBoundingBox(vertices)
// The highest value of the mask (between 0 and 1) in the range of the boundingBox.
highestValue = HighestValueInMask(boundingBox)

// The diagonal size of the boundingBox, this is what we will use as comparison.
boxSize = magnitude(boundingBox.topLeft, boundingBox.bottomRight)
// The operation will give us the minimal size this face can have.
currentValue = (minSize - maxSize) * highestValue + maxSize;

// If the size of the boundingBox is higher than the value, it means that we still have
to subdivide.
return boxSize > currentValue
```

By applying this formula to all faces and then all to all subdivide faces we thus recursviely apply the algorithm to the whole mesh.

Here is an example of an original single square that is divided based on the influence layer as seen to the right. We can see that the subdivsions in the image to the left follow the pixels becoming lighter as we head higher up.

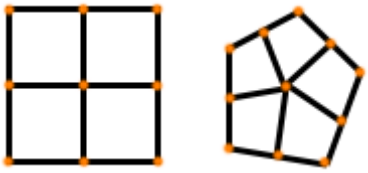


Center Point

To apply our division algorithm, we must first of all find the center of the polygon. To do this, we simply find the average of all the vertices.

Let's suppose the following faces with their orange vertices and black edges.

Once we found their center and divided they will look like this :



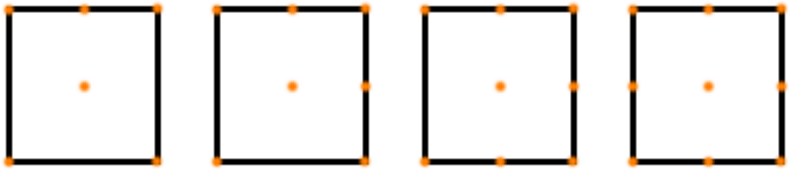
Divide the face

Now that we have the polygon's center we can subdivide the face.

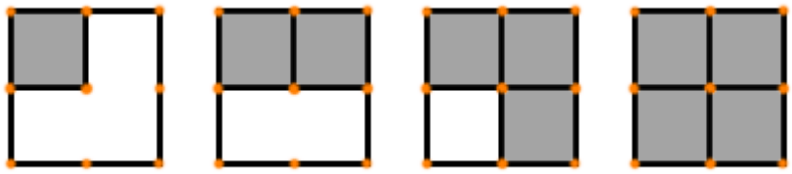
We'll start with a standard square :

We find the center point :

Then we add vertices by dividing the edges in two :



Finally, we build our new faces with our added vertices.



The original face is removed from our *queue* data structure and the new faces are added.

It should be noted that a divided face will always return N new faces where N = the original number of vertices.

Finding neighbours

Once we figured out how to subdivide a face, we need to answer the following question : What happens to the neighbouring faces when we split an edge?

We would want to ensure that the split is shared between the two faces. The solution for this was to keep a vector split edges and have faces search for them while they subdivide. So we have the data structure "*std::vector<SplitEdge> splitEdges*" to keep care of that.

- **SplitEdge** contains three indices, the two indices of the original edge and the new indice that split them apart.

So we introduce new code in our algorithm :

1. If we find that an edge has been split, we don't split it ourselves to retrieve the **SplitEdge** object and apply that split to our face.
2. When we split an edge, we look for a face in the ***completedFaces*** stucture that has the same edge. If we find such a face, we remove it from the ***completedFaces*** structure and add it back to the ***toProcess*** structure.
3. When it has been decided that a face will no longer be devided search for any of it's edges that might have been split. If so, we introduce the splits into the face and proceed by adding the face to the ***completedFaces*** structure.

```
// If our edge has already been split we will find it
EdgeSplit split = FindEdgeSplit(vertex, vertex + 1)
if split != null:
        return split.SplitPoint
else:
        center = CenterPoint(vertex, vertex + 1)
        split = EdgeSplit(vertex, vertex + 1, center)
        // Keep a vector of all split edges to be found
        allEdges.Add(split)

        // Find a face that shares the same edge and return it to the queue
        MeshFace face = FindFaceWithEdge(split)
        if face != null:
                completedFaces.Remove(face)
                toProcess.Add(face)

        return center
```

The steps combined

Now that we have all the steps figured out, we can combine them all :

```
class SplitEdge:
        start, end, center

// Filled with all the original faces
queue toProcess = BuildFaces()
list splitEdges = {}
list completedFaces = {}

while !toProcess.empty:
        current = toProcess.pop()

        if current.shouldSubdivide:
                // This method will look through splitEdges to find split edges. And
then update the face if needed.
                UpdateFaceWithDividedFaces(current, splitEdges)
                completedFaces.add(current)
                continue

        forall vertex in current.vertices:
                edge = FindEdge(vertex, vertex + 1)
                center = null
                if edge == null:
                        center = Mid(vertex, vertex + 1)

                        edge = SplitEdge(vertex, vertex + 1, center)
                        splitEdges.add(edge)

                        // This method will look through completedFaces to find a face
that has the same edge as the one that has been split. If found, it is removed from
completedFaces and added to toProcess
                        ReturnCompletedFaceIfSplit(edge, completedFaces, toProcess)
                else:
                        center = edge.center

                face.AddBetween(vertex, vertex + 1, center)

        forall vertex in current.vertices:
                toProcess.add(Face(vertex, vertex + 1, vertex + 2, current.center)
```

This pseudo code is mostrly usefull as a quick explanation, for a better comprehension it is best to read the actual code : influenceMesh.cpp.

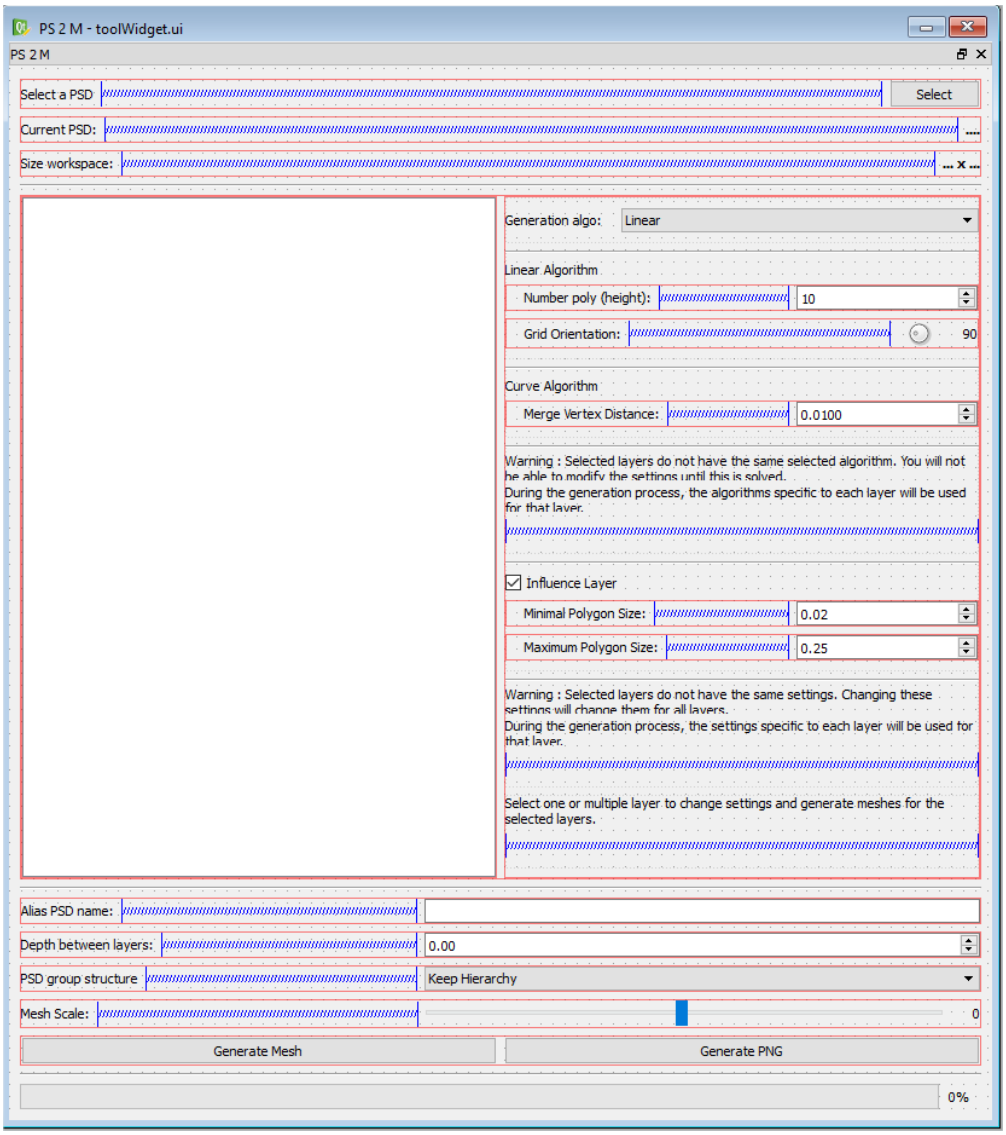
UI

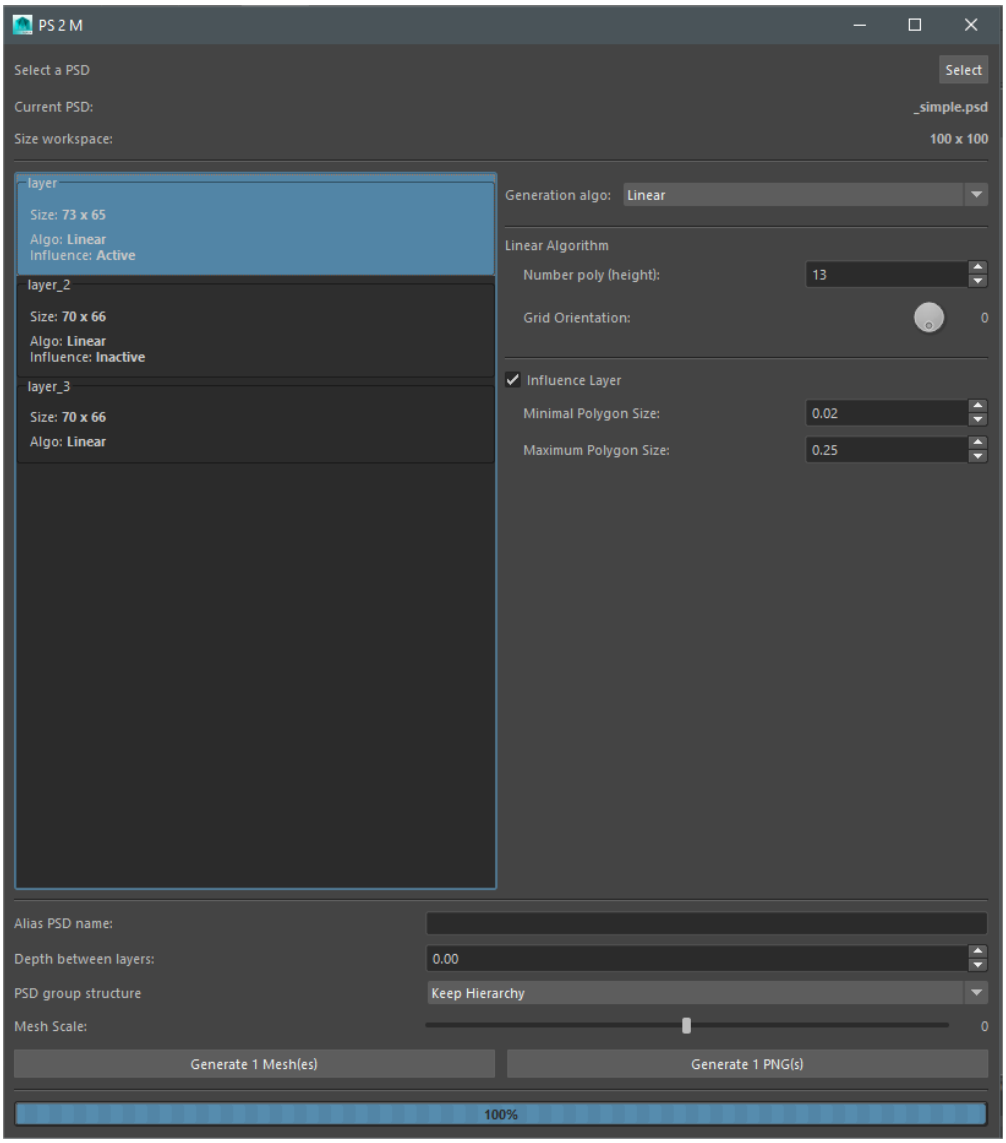
Introduction

This tool delivered conjointly with an interface that can be opened with Maya, the 3D software. This interface is built with Qt Designer, a software dedicated to ui and interfaces.

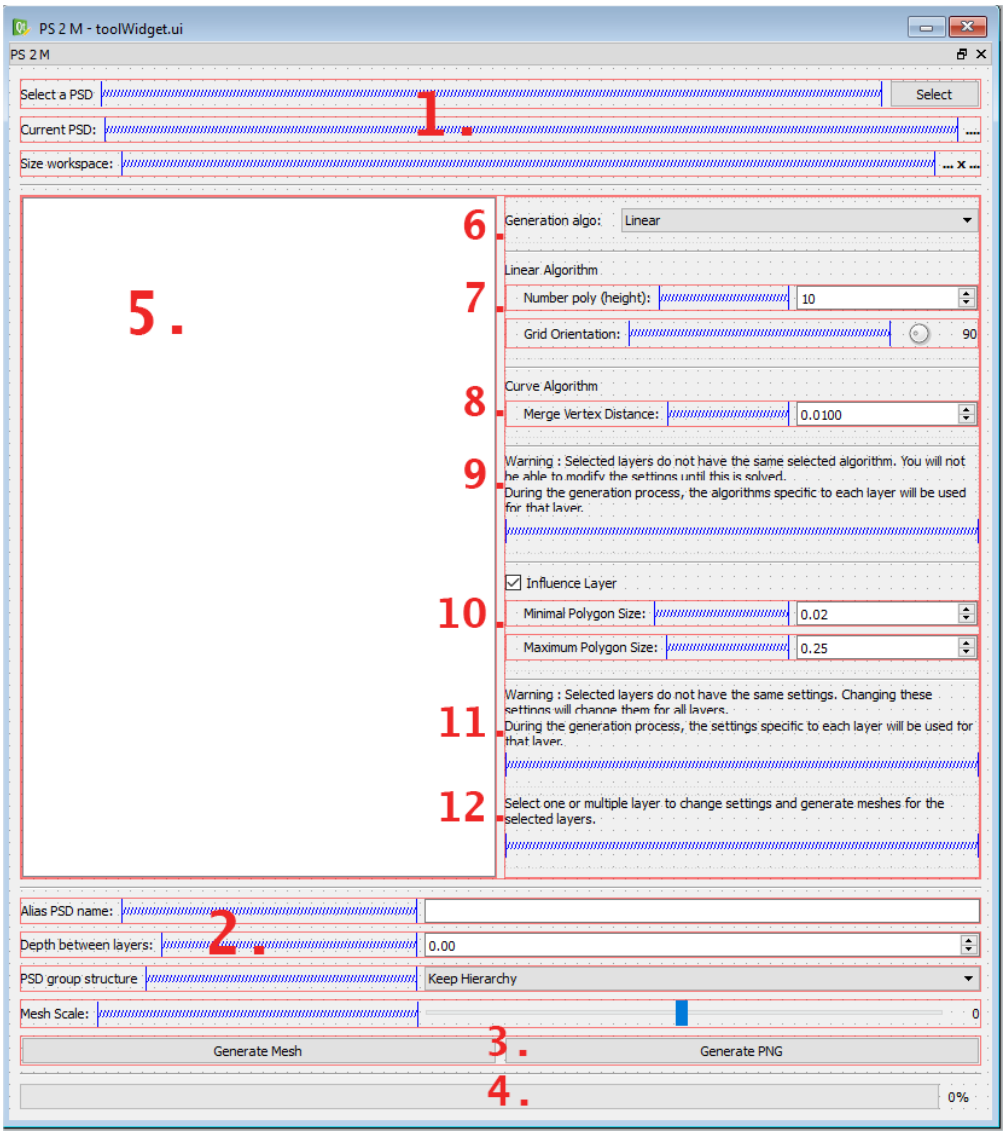
The interface

Inside Qt Designer we work with an interface which looks like the first image, and once in Maya it looks more like the second dark image:





Here is an explanation of the interface in Qt Designer :



1. This section is the Header and allows you to select and display information about a PSD file.
2. The parameters completely to the bottom of the interface are global to all layers.
    a. "**Alias PSD name**" : This is the name that will replace the PSD file's name in Maya. If this value is not empty, the root of the hierarchy in Maya will be equal to this name, otherwise the PSD file's name will be used.
    b. "**Depth between layers**" : The distance in Maya units between each generated Mesh.
    c. "**PSD group structure**" : Allows you to choose between keeping the full hierarchy in the PSD file or to flatten the hierarchy in Maya.
    d. "**Mesh Scale**" : A multiplier for the size of the generated meshes. (At 0, the size is multiplied by 1, at 0.5, the size is multiplied by 1.5.)
3. These buttons allow you to generate the meshes or PNG images.
    a. These buttons will updated based on how many layers are selected in section (5). They'll only generate what is selected, or if **none are selected, they will generate all layers.**
4. The progression bar will show the progress of different tasks.
5. This section contains the list of layers that come from the PSD file. We can also see here information about the layer and its parameters.

    a. Selecting one or more elements from this list will update the section to the right of this listé. The sections (6) to (12) will added or removed depending on the parameters of this selection.
    b. The modification of the selection will also update the buttons in the section (3).
6. This section allows you to modify the algorithm used to generate the mesh.
    a. If "**Linear**" is selected, the section (7) will appear while the section (8) will disapear.
    b. If "**Curve**" is selected, the section (8) will appear while the section (7) will disapear.
7. This section contains the parameters used with the "**Linear**" algorithm.
    a. "**Number poly (height)"** : This value is decides the number of polygons (in height) used in the mesh.
    b. "**Grid Orientation**" : This value decides the orientation of the generated mesh. To note, the values 0 and 180 will produce the same results.
8. This section contains the parameters used with the "**Curve**" algorithm.
    a. "**Merge Vertex Distance**" : This value is used to combine Vertices that are too close to each other.
9. This section contains a warning that explains that the selection from section (5) contains layers with different algoritms.
    a. If this section is visible, the sections (7) and (8) will be removed.
10. This section contains the parameters used during the "**Influence**" algorithm.
    a. This section appears only if at least one of the layers as an influence layer associated with it.
    b. This section can be deactivated, and if deactivated, the "Influence" layer will not be applied.
    c. **"Minimal Polygon Size**" & "**Maximal Polygon Size**" : These two values represent the size of the polygons related respectively to the white and black pixels of a mask (the influence layer).
11. This section contains a warning that explains that the selected layers do not share the same parameters.
    a. If this section is visible, the sections (7), (8) and (10) will correspond to the first layer selected.
    b. If a parameter is modified, the parameter will be modified for all selected layers.
12. This section contains a warning that says that no layers are selected.
    a. If this section is visible, no other section will be visible.

Parameter Serialization

The parameters of the PSD file and its layers are serialized and saved in a text file. This file is in a JSON format and is saved in a folder that accompanies the PSD.

1. The JSON format has been chosen for its simplicity, ease of use and how easy it ise to find a simple library that serializes it.
    a. Also, an extra advantage of JSON (versus binary serialization) is that the information is human readable/ modifiable.
2. When a PSD file is selected in the interface a new folder is created with the same name and same repertory as the PSD file.
    a. This folder is used for both the serialization file and for the PNG export.

The serialization of the parameters in not completed automatically, we must code which values are serialized or not. When we want to add a new value in the save, we must :

1. Add the value in the method **GlobalParameters::DeserializeContents** in the corresponding area. This method receives a JSON object in parameter and assigns the parameters based on the JSON.
2. Add the value in the method **GlobalParameters::SerializeContents** in the corresponding area. This mehtod creates and returns a JSON object created with the paramters.
3. Also, if the added value is global, it will be possible to add it to **GlobalParameters::SetDefaultValues**.

The sections

To show and hide the different sections of the interface some logic must be applied for each. This logic can be found in the method **ToolWidget::UpdatePanels**.

- Generation Algo (6) : If there is at least a single layer selected.
- Linear Algo (7) : If all selected layers are of the Linear type.
- Curve Algo (8) : If all selected layers are of the Curve type.
- Multiple Algo (9) : If there are more than one selected algorithme type.
- Influence Layer (10) : If there is at least a single layer with an influence layer selected.
- Paramètres Multiple (11) : If there is at least a single layer selected and that there are shared values that are different.
- Paramètres Vide (12) : If there are no layers selected.

Shared values

A warning message is shown when share values are not equal. This allows the used to know that when they modify a field, all selected layers will also be modified. If no changes are made, the values will stay as they were before the selection. Also, during the generation, the values of each layer will be used and not necessarily those shown.

Finding if the shared values are different is done automatically. This process is done in the method **ToolWidget::AreSelectedValuesDifferent**. So when a new value is added to the parameters, it should also be added in this method so that it is considered in the process.

The layers

Each layer in the section (5) has a description, here are it's contents :

- The name of the layer.
- The layer's size.
- The algorithme that will be used for this layer.
- If the layer has an influence layer associated, and if it's activated.

The following methods are available if the description needs to be modified : **ToolWidget::ApplyLayerDescription** et **LayerParameters::UpdateDescription**.

PSD PARSER

Introduction

The creation of PSD's "parser" is based on Adobe's documentation of the file format.

- https://www.adobe.com/devnet-apps/photoshop/fileformatashtml/

Document specification presentation

The PSD reader manage transcribing the content in a data format in C ++, making it possible to use this data in another C ++ module.

The interest in the project was to go retrieve the following data:

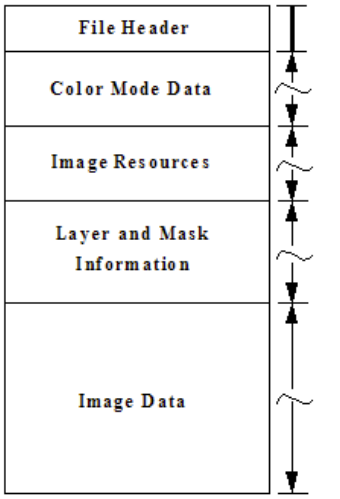
- Dimension of the work area.
- The information of "General path".
- The information of each layer, dimensions / name / effects.
- The textures of each independent layer.
- The vector mask associated to the layer.
- The Group Hierarchy for layers.

We used the following documentation to read and construct the data structure.

- https://www.adobe.com/devnet-apps/photoshop/fileformatashtml/

Here is the composition of a PSD file, we have tried to keep similar names for the data structures intended to receive the content.

Photoshop file structure



"File Header"

- The basic properties of the PSD, dimension.

"Image Resources"

This resource space contains all the psd information related to general photoshop operations. They are recorded by data block, each one being different.

As part of our project we only read structures that are useful to us, the rest is not preserved.

- The general Paths information.
  - L'id for the paths is : **2000-2997.**
- The information of the resolution.

"Layer & Mask Information"

The information of the layers is split into two parts:

- A list of all the properties of each of the layers.
  - The information of each layer, dimensions / name / effects.
  - The masks of the vectors associated with the layer.
  - The Group Hierarchy for layers.
    - By default each entry in the "Layers" section of photoshop is a layer.
    - A layer is by default a texture.
    - Otherwise a standard value is assigned to it to specify whether it is Group open or closed (Folder containing layers).
- A list of textures ordered by layer and separated by channel. So for each layer the writing is done by channel, according to the order ARGB.
  - The textures of each layer are independent.
  - Different formats can be saved, we process Raw and RLE.

> Reading Layer & Mask Information offers several possibilities. Indeed the layers can be contained in the section "Layers records" in 70% of the cases. Exceptions happen and the "Layers records" section is empty. The information is then stored in the same way (structure "Layer info" + "Channel image data") but in an "Additional Layer Information" introduced by the ID "Lr16".
> So if "Layer info" has a size of 0, check the size of "Global layer mask info" then go to "Additional Layer Information" everything is in an ID.

"Image Data"

- This section contains the data of the merged texture of each layer, it is in the same format as that of the layers.
- No interest in the project to read this section, it is not kept.

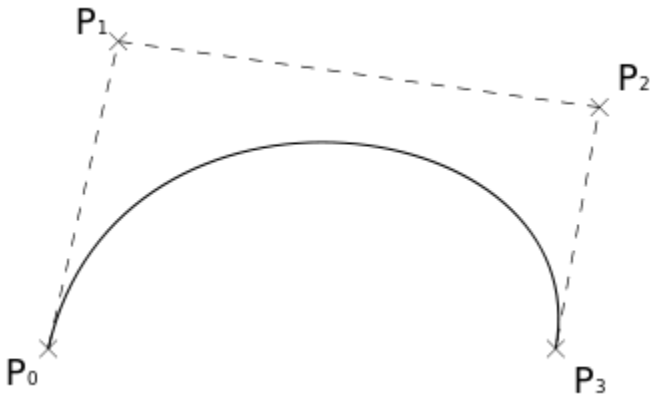
Dependency with "Util"

The project uses the notion of bezier curve found in Photoshop. It's curves are archived in the PSD according to the concept of "closed" or "open" curve. Bezier curves are actually a mathematical function to draw a series of points based on anchor points. These are his points that are saved in the Photoshop file and that we use in this module to rebuild the curves.

The advantage of the bezier curves lies in the fact that regardless of the resolution, each stroke is not stored as a pixel sequence, but as a value obtained by calculation.

We are talking about a cubic bezier curve in our case:

- https://fr.wikipedia.org/wiki/Courbe_de_Bézier



"Util" therefore contains the storage structure of a point. In our example:

- **P1** is the influence exit point of ***P0.***
- **P2** is the point of entry of influence of ***P3.***
- **P0** and **P3** are inking points, ie the points used for drawing the curve.
- **P1** and **P2** are points of influence of the path according to the mathematical formula.

For a point P there is always a point P1 and P2 associated with this point. That's what we keep. If the points P1 and P2 are not apparent is that they have the same value as their inking point. We find the use of this data structure in the project "Mesh generator" to calculate the curves and find the intersections.

References

Here is a list of useful links for designing this PSD player.

- https://www.adobe.com/devnet-apps/photoshop/fileformatashtml/
- http://telegraphics.com.au/svn/psdparse/trunk

EXPORT PNG

Introduction

As part of the project we wanted to export the textures of each layer in a convenient format for Maya.
We tried some format like the "iff", an uncompressed format that maya integrates.

The needs of use have brought us back to our initial choice PNG. A compressed format that handles the alpha channel very well for transparency.

Our first attempt

The iff format had the advantage of being able to load the texture read directly from the PSD directly into the Maya texture manager.

- No need for compression library.
- Less texture transformation operation.
- Directly linked to the destination tool, MAYA.

Maya functions were used to record the iff image on disk. This uncompressed format created large files knowing that each pixel value was written.

in Mayan use refused to load more than 5 textures of 4096 by 4096 simultaneously.

It was from this point that we looked at the PNG format.

Information on PNG

Png is a convenient compression format, especially for the quality of the image but also for its management of the alpha channel useful in animation to maintain transparency.

The following documentation has information on the PNG format:

- https://fr.wikipedia.org/wiki/Portable_Network_Graphics

As part of the project we read the texture either in RAW or in the RLE Compression Format, then we store it so that we can prepare it to a specific format for PNG compression.

We store 4 tables containing the values of each pixel per RGBA channel. When we give it to the PNG library, we transform this data into a pixel array where the values of each channel for one pixel follow each other in RGBA order.

Choice of the library

We have considered integrating the Libpng library into the project, it is a practical library but quite important.

- http://www.libpng.org/pub/png/libpng.html

Therefore we have oriented our choice to Lodepng with several advantages.

- https://lodev.org/lodepng/
- She has a simple script.
- No dependency on a compression library.

Generation Of Textures

One of the advantages of doing the texture generation via the plugin in maya is the speed of compression that we have when we use the built-in script.

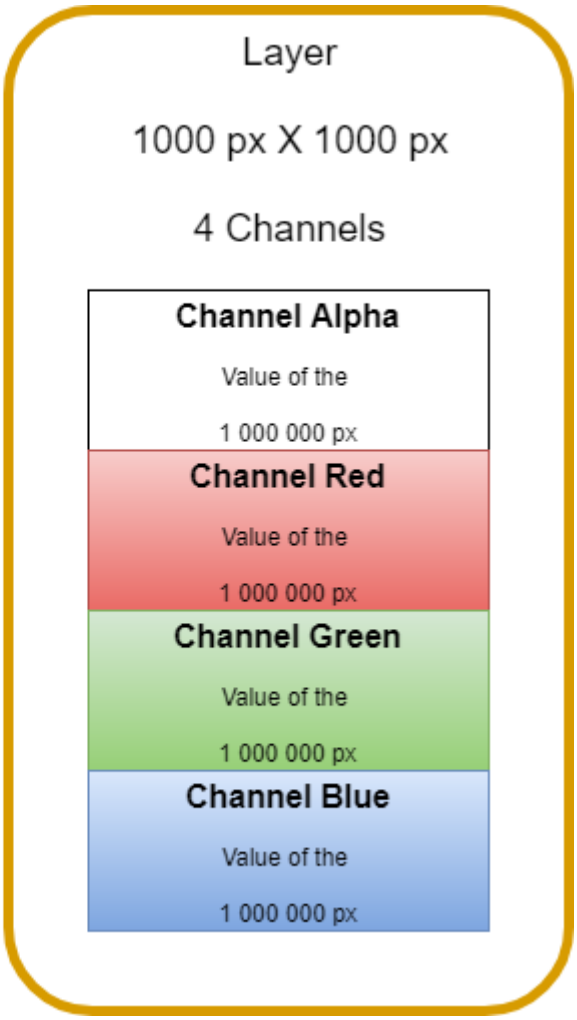
Indeed the use of scripting compression in Photoshop is much longer. The compression format proposed by lodepng is equivalent to the largest compression format proposed by Photoshop for Png compression.

The use of script:

- Simplifies the integration of the functionality into the project.
- Centralize operations on the project.
- Significantly increases the export time of layers.

Data Conversion

An archive psd for each layer all values of one channel at a time is for 4 channels we have the data as follows:



To allow the Png Converter to understand the data we modify the texture preserved by writing the information as follows:

- Texture = **Layer * Value** of one pixel.
- Value of one pixel = value **RGBA** concatenated.

Photoshop retains its cropped layers based on alpha but also compressed in RLE format.

When reading the layer, we save it cropped but in Raw format. Cropping allows us to save memory. Raw does not have compression applied.

To answer the need of the project we transform the texture to the size of the working area of the PSD by filling in the values of the missing pixels by 4 bytes of value 0, ie one pixel of alpha, so we will have UVs corresponding visually to our mesh. Then we apply Png compression before saving it using LodePng.

MAYA COMPONENTS (EDITOR AND DATA)

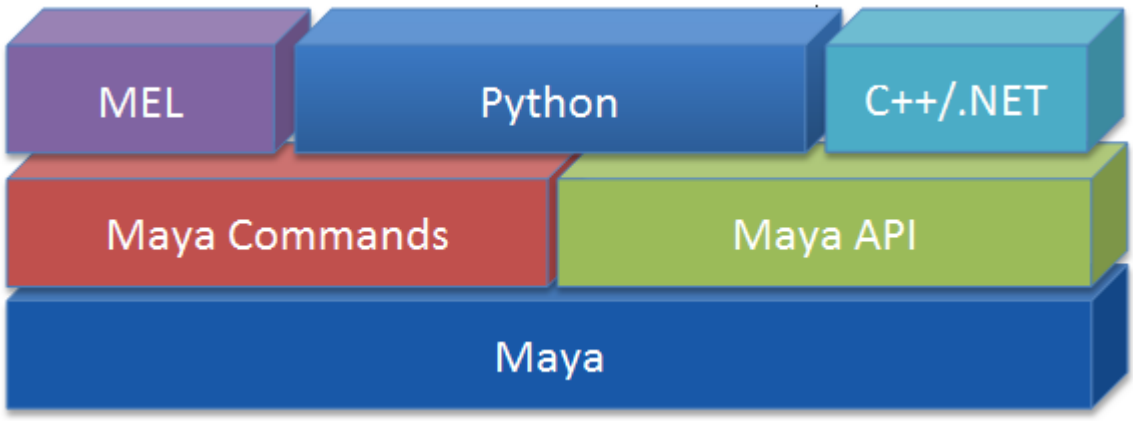
Introduction

This section represents an important part of the project, it manages the loading of the plugin in maya, the commands, the link between the reading of the PSD and the generation algorithms, the UI and its parameters. But above all, it creates all the components that will allow Maya to display a mesh and its texture.
Maya exposes many options via the API, but finding out how the heart of Maya works takes time to adapt.

This documentation tries to give you the essentials notions to navigate among the functions of maya.

Details of scripting options

Maya is composed of several layers allowing access to different internal resources.



We take the decision to develop the plugin with the Maya API in C ++. The following documentation specify that the use of the API offers a real performance gain in the execution time of operations.

- https://help.autodesk.com/view/MAYAUL/2018/ENU/?guid=__files_API_Introduction_htm

As part of the project there are some objects created via the API by a call from a "Maya Command".

The command call goes through the Global class, the main architecture of the API.

Example:

- "MGlobal::executeCommand("ls -sets", result);"

The different objects of the API

C ++ Object Types

- Maya objects (**MObject**)
  - All is a MObject (curve, surface, Dag nodes, shader, ...).
  - These are just pointers to different sub-components.
  - Do not keep Reference on a MObject, the use of iterators is recommended when looking for a more precise object.
- Function sets (**MFn**…)
  - These objects are actually objects that allow operations on an object of the type corresponding to what follows the "**MFn**".
  - Example **MFnMesh** allows you to do operations on the mesh.
  - They are important because they operate on the object's data structures, while doing more general service operations to maya.
  - **MFnMesh** modifies or creates the mesh data associated with an identified object, while referencing this data to the Mayan Geometry management system.
- Proxy objects (**MPx**…)
  - The type of object that must be derived to create a new type of object recognized and managed by maya.
  - They are objects containing the data.
  - Adding an command requires inheriting **MPxCommand**, so Maya will add this object to the command management system.
- Wrappers (**M**…)
  - **Wrappers** are classes of operations often on properties independent of maya objects or general systems of maya.
  - Example of mathematical operation classes.
- Iterators (**MIt**…)
  - These classes are iterators to the different systems of maya and objects managed by these same systems.
  - Example There is an iterator on the mesh geometry, there is one to the editor nodes.
  - These are frequently used objects to retrieve the contents of an object and apply operations to it.

Useful Classes

- **MPxCommand**
  - Deriving from this class allows you to create a new command recognized by maya.
  - A structure is used to call the operation execution.
- **MGlobal**
  - Master class of Maya, contains an important list of high level operation.
  - Log display.
  - Object removal at the editor level with the referenced data in the centralized systems.
- **MDagPath**
  - Reference to the DAG (Acyclic Dependency Graph) object, which is the publisher node for Maya objects.
- **MSelectionList**
  - Stores a list of components present in a scene.
- **MItDag**/**MItDependencyGraph**
  - Both are useful for navigating editor nodes and different connections between objects.
  - The **MitDependencyGraph** class makes it easy to go from node to node.

- **MPlug**
  - Represents the inputs and outputs of an editor node.
  - Convenient to manage the connections between the different components.
- **MDGModifier** / **MDagModifier**
  - **MDGModifier** is used to perform operations on the visual part of maya objects (editor node).
  - It manages the creation and modification of nodes by applying an operation list on an object. **Undo / redo** management.
  - **MDagModifier** has the same functions but more specifically for the data contained in the objects.

General Systems Management

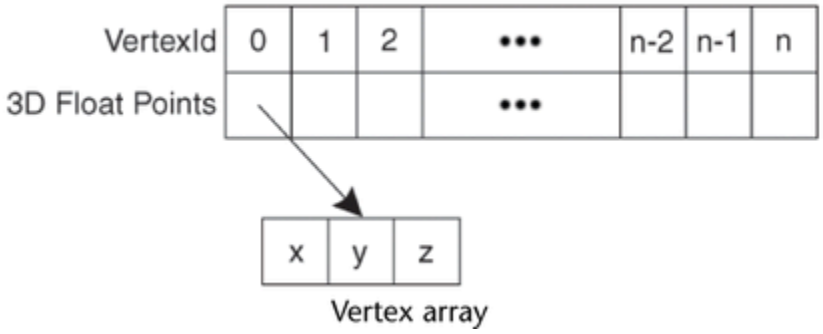
Maya incorporates many implicit and inaccessible manager. Shader management, geometry management, textures and nodes. Do not hesitate to use the iterators to go through the entire project and refine this research and evaluate the "**kType**" objects and reduce the spectrum of work.

It is common to check if properties are existing to select objects.

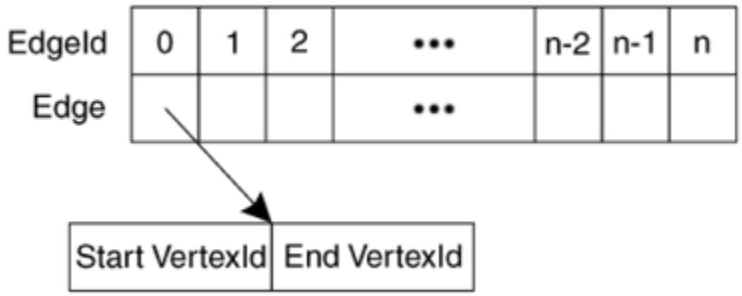
Creating a mesh and its components

Here is the structure of a mesh as illustrated by the maya documentation.

The vertices are stored as a table where the values are saved as an array of 3 floats. The vertex **index** is important because it will be used as a reference in the construction of polygons.



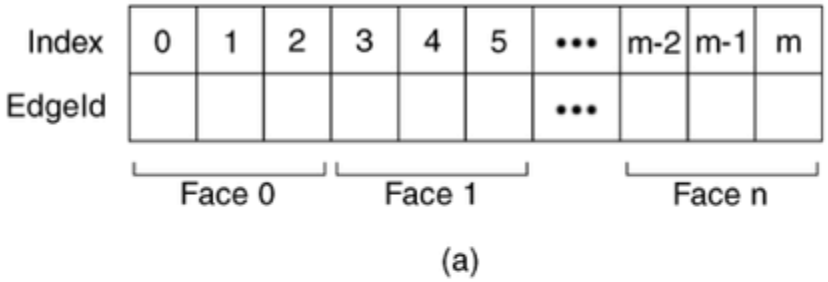
The edge specification allows you to manipulate the link between two points by specifying the index in the previous table of the starting point and the arrival point.



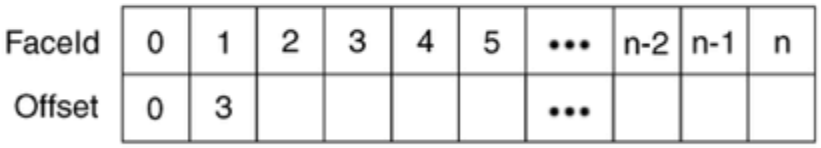
Edge array and edge structure

The construction of the mesh is done by filling two tables, the first table where we adds for each face the list of the involved edges ordered in the same direction for each polygon.

The second table specifies the reading of the first table, keeping for each index the number of stops to read in the preceding table. For a given point the sum of the previous values gives the offset.



(a)

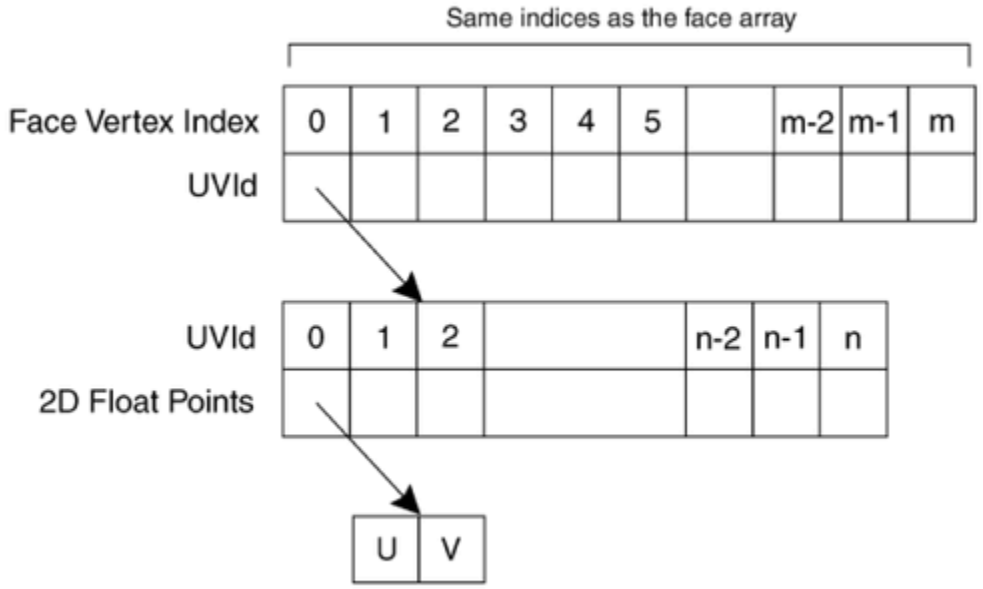


(b)

Face array and face index array structure

UV

UV management is like flattening points on a texture. For each vertex have specified the coordinates on the texture. It is therefore an association vertex ID (x, y, z) / coordinate 2D (x, y).



UV array and UV index array

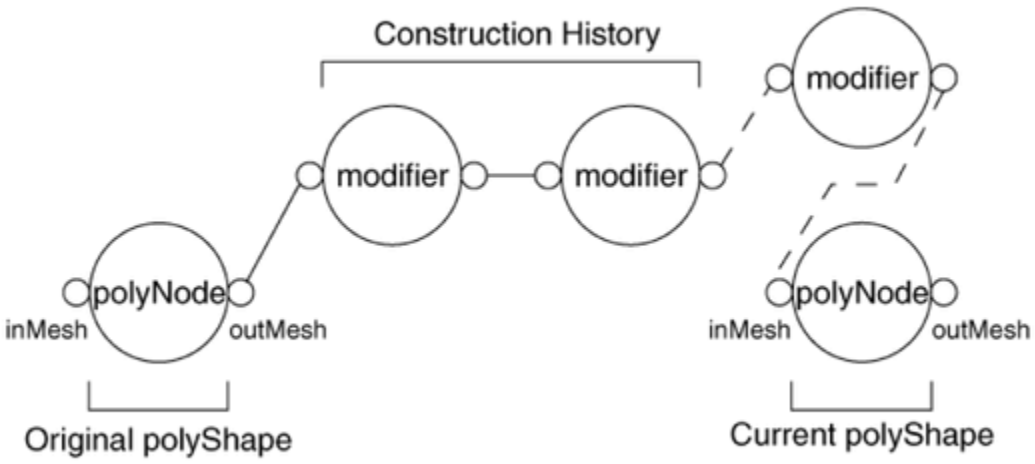
Modifier Et Operation

The operation of creating a mesh is in two steps, the first is to use an MFnMEsh object to create an object containing the type "**kMesh**".

MFnMesh allows to do the operations on the geometry of the mesh, but also on the UVs.

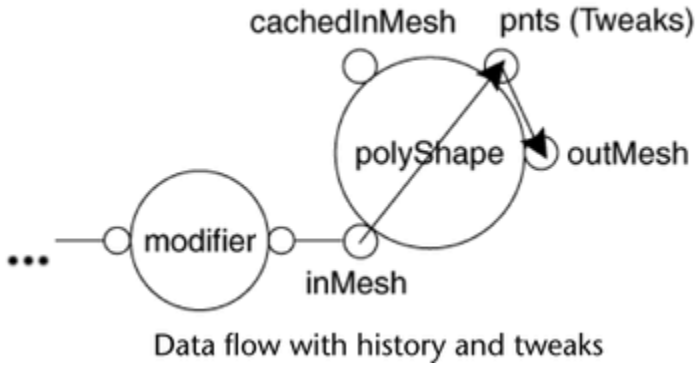
Subsequently, for its operations to be effective and recorded for an object with the different systems referencing it, it is necessary to use an object of the "**modify**" type such as the **MDAGModifier**.

The "Edit DAG" records the operations, then when calling the "**doIt** ()" function, the actions are applied, the history of operations on the object is updated to allow the undo / redo. It is at this point that the nodes become apparent in the "system DAG".



Inserting a modifier node

It is important to understand that the "**Edit**" retains the series of operation and injects it into the object, the history retained will be the operation and the resulting change will be the data stored in the object as well as the difference between before and after the operation.



Data flow with history and tweaks

The "plungin command"

In the project we have only one command implemented. It manages the opening of the plugin menu. It is necessary to implement as much command as action related to the plugin. The definition of a command operation is done in the call of the "doIt ()". The class must inherit MPxCommand so Maya can subscribe to its system the new commands created.

In our case, the Qt interface handles all the operations.

It must provide a command by operation.

There are two methods for managing the subscription of system commands, "**InitializePlugin** ()" and "**UnitializePugin** ()". The commands are then added in maya when the plugin is initialized

and removed when the plugin is unloaded. These commands are considered as the main execution during loading and unloading ("Main").

The integration of the UI

Maya offers Qt integration to easily define a graphical interface with the Mayan style. Each version of maya has an edit version of **Qt**. Maya 2016 uses qt **4.8.6** and maya 2017/2018 use version **5.6.1**.

The "dev kit" contains all the references for the compilation. In our case the UI is a singleton to have only one instance of our menu.

As a result, the last loaded PSD is retained even when the window is closed.

PLUGIN PHOTOSHOP AND ADOBE CSXS

Introduction

Creating a plugin for photoshop requires to know two different systems.
The first concerns Photoshop scripting actions, more precisely the "actions" that can be done in javascript to automate operations using the native photoshop code.

The second concerns the extension management system which include the UI aspect, the management aspect of the installation and the uninstallation of the extension, as well as the calls of the actions mentioned above.

Development environment

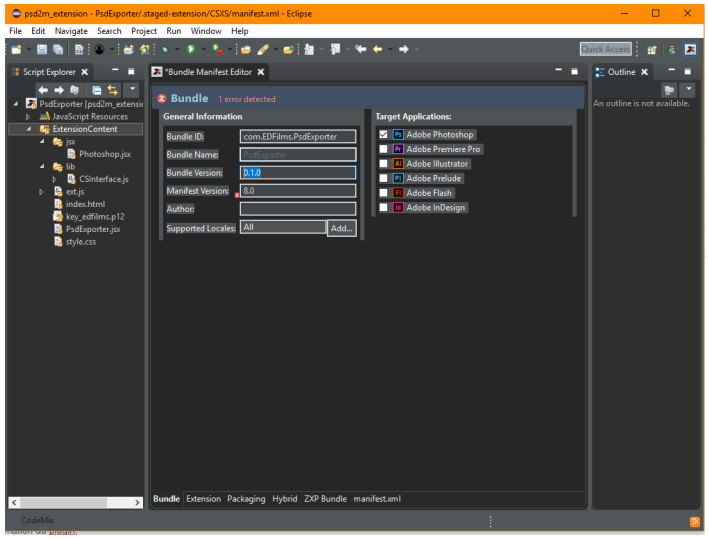
There is a project creation "**template**" here:

- https://labs.adobe.com/downloads/extensionbuilder3.html

This "template" offers a file structure as well as the basic implementation of scripts useful for defining an extension. The template allows you to create plugins for the entire adobe creative suite.

One of the important points is the management of the "manifest". It is an XML file for setting the extension information as well as the versions and tools of the Creative Suite that can use it.

- Photoshop version management
- Version management of the supported extension system.



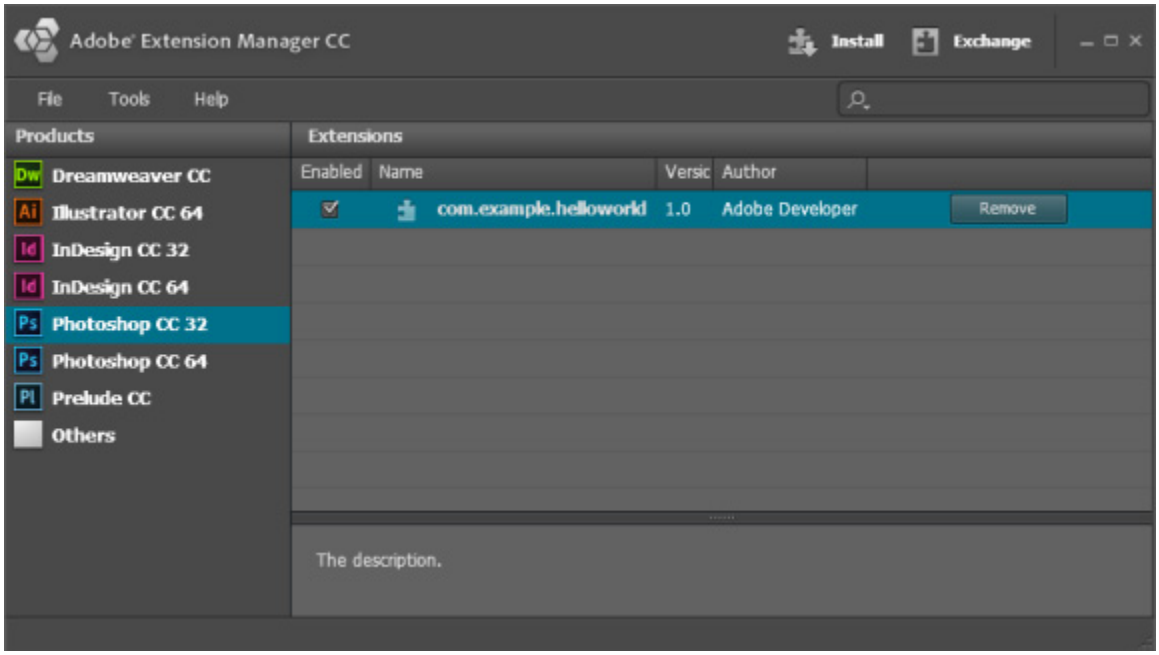
The structure of the project presented in the previous illustration is as follows:

- "index.html" and"style.css"→ Used to define the interface of the extension.

- "ext.js" → Script that operates the link between the UI and the executed operations (Actions).
- "jsx/Photoshop.jsx" → Contains the definition of the operations executed on each layer.

Adobe CSXS

Adobe offers an embedded module with the creative suite. This CSXS module is actually the manager of the extensions managed on your machine. The following utility allows you to view the extensions present for each adobe product.



ZXP allows you to install extensions, that's what we use for project extension. It is embedded in the plugin installer.

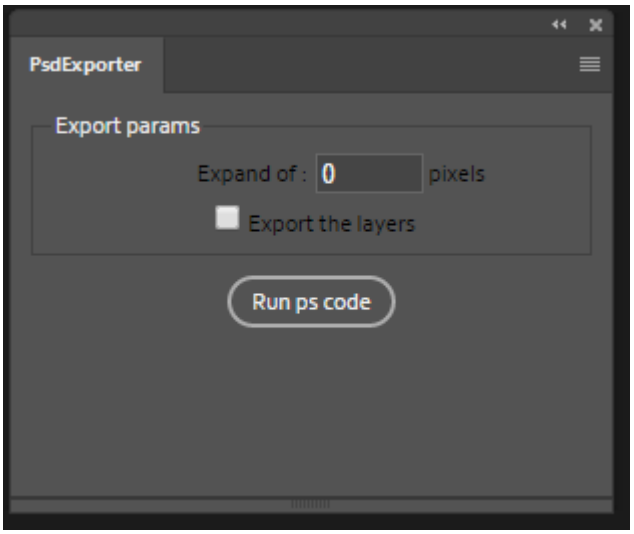
- https://helpx.adobe.com/ca_fr/animate/kb/install-animate-extensions.html

Procedure
:

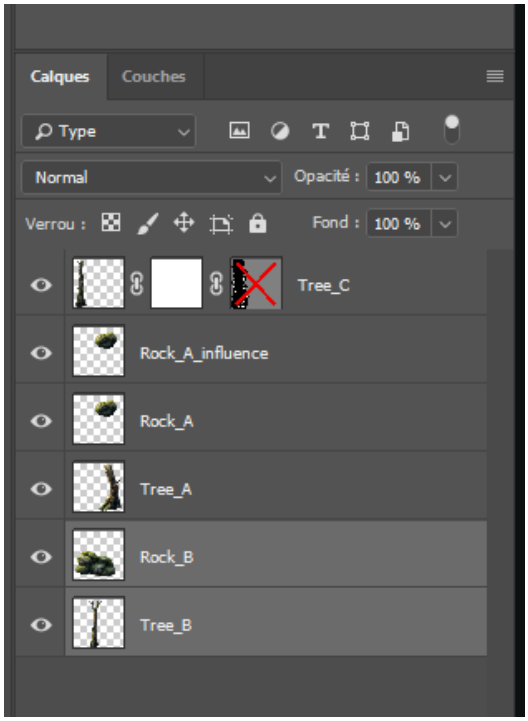
1. Téléchargez the ExManCmd command-line tool for Windows.
2. Extract the .zip file to a location on your desktop, such as / Users \ [UserName] \ Desktop \
3. Copy and paste the extension (.zxp file) that you want to install into the same folder (C: \ Users \ [Username] \ Desktop \ ExManCmd_win \) to find it more easily.
4. Open the command prompt by selecting Start> All Programs> Accessories> Command Prompt.
5. Enter "cd**"** and fill in the path to access the extracted folder:
    a. cd C:\IUsers\[UserName]\Desktop\ExManCmd_win
6. Once in the ExManCmd_win directory, enter the following command at the command prompt and press "Enter":
    a. ExManCmd.exe /install “SampleCreateJSPlatform.zxp”
7. **Note**: If the extension name contains spaces, enclose them in quotation marks ("").

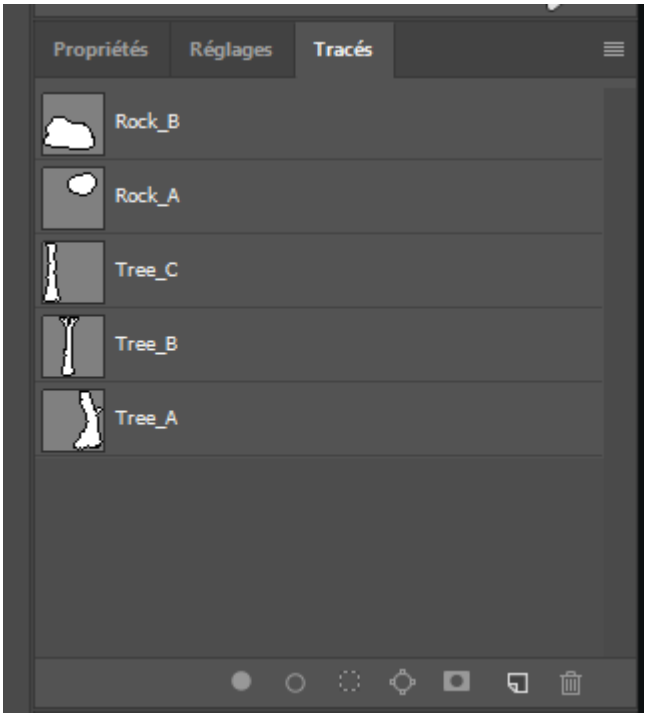
Plugin operations

Here is the list of operations that occur when you use the psd export plugin.



- Specifying a copy folder of the PSD to perform operations on a copy, allows the user to keep his work.
- Rasterization of layers.
- Deletion of "general layout" layers, selection and expansion of the contours of each layer according to the parameter specified in the "**expand of**" interface.
- Generation of a path layer for each layer according to the selection made in the previous step.
    - We use the name of the layer to preserve the link between layer and layer of path.





- Save changes.

- PNG export is available but very slow. "**export the layers**"
    - Indeed the call by the layer of scripting of photoshop does not allow to use directly its API, which causes a slowing down of the treatment. It is best to go through Maya for this step.

After studying the different possibilities for this plugin, implementing an extension from the adobe C ++ API seemed very difficult and uncertain. Having little documentation on the subject, we were not able to guarantee a result.

Adobe offers a high-level script or action set to interact with the layers in the canvas.

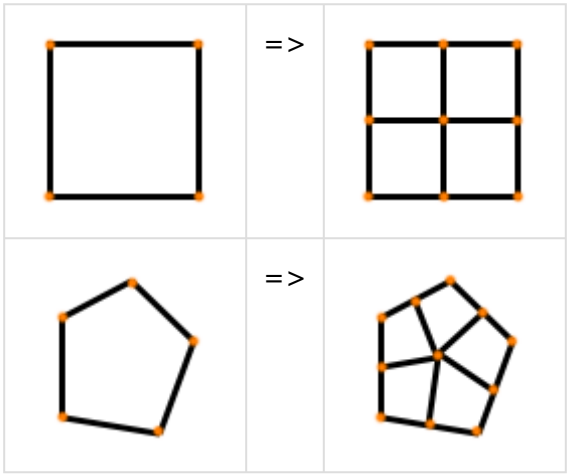
So we combined in this extension the scripting (actions) and the creation of a UI.

POLYGON SUBDIVISION

Introduction

One of the stepf of the Influence Algorithm is to divide faces of a mesh based on the concept of Subdivision Surface. Each division will add a number of faces equal to the number of vertices of the original face.

The following simple examples show the results of a face subdivsion.



To complete a subdivision we first needed to solve two problems :

1. Find the center of the polygon.
2. Divide the edges of the polygon.

1. The centre

The first method that was considered was to average all the vertices of the polygon.

Advantage

- The average of points is an easy and quick calculation.
- For most cases, the average is largely sufficient to approximate a centre point.
    - For all symmetric polygons, the average **is** the center.

Disadvantage

- If the polygon is asymmetric, we could see a sway of the center towards "heavier" side of the polygon.

- If the polygon is concave, we could end up with a center point that is outside the polygon.

Solution To The Disadvantages?

Instead of looking for a center, we could look for a centroid. A centroid is a point that is at the center of gravity (or visual center).

Some options are available to solve this problem :

- https://www.mathopenref.com/polygonirregular.html
  - This solution uses the area of the polygon.
- https://blog.mapbox.com/a-new-algorithm-for-finding-a-visual-center-of-a-polygon-7c77e6492fbc
  - This solution uses a fractal formula to find the center.

Final Decision?

We decided to use the average of the vertices

- For the needs of this project, a higher precision offered by a "better" solution would not have added any plus-value.
- The algorithms used to find a centroid add a large amount of calculations (versus just using the average), and would increase the overall process time as these algorithms would need to be applied to each face.

- Even though the Influence Algorithm could be applied to any mesh with possibly any type of face, in this project it is only applied to meshes that have been created previously by our own algorithms. So we are ensured a minimal "regularity" of the polygons.

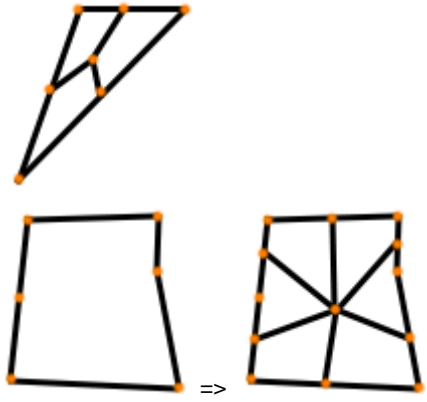
2. The division

The first process envisioned was was to split an edge in two perfect halves and place a new vertice at that center point.

Advantage

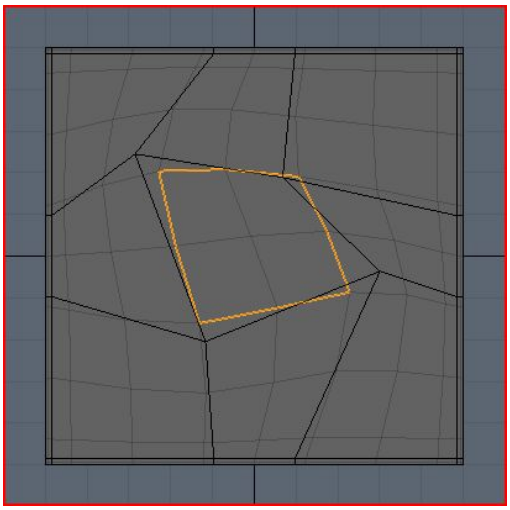
- Fast calculations
- This technique works perfectly for regular polygons.

Disadvantage

- A narrow or deformed face (when compared to a square) would result in narrow or deformed faces.
- 

Solution To The Disadvantage?

The Catmull-Clark algorithm is a technique to divide a mesh into smooth faces. Often used in 3D, this solution could be used during the subdivision process to "straighten" the faces.



This image was taken from : http://www.rorydriscoll.com/2008/08/01/catmull-clark-subdivision-the-basics/. An explanation of the algorithm can be found at this same web page.

Final Decision?

The Catmull-Clark algorithm was found while this documentation was being written, at the end of the project's life. While looking for articles we ended on the explanation of this algorithm. If we had found it earlier we most probably would have included it in the project to smooth the faces. At a minimum it could of been an option to the user.

Bu then again, the curretn algorithm perfectly follows the demands of the project, anything else would of been of extra value.

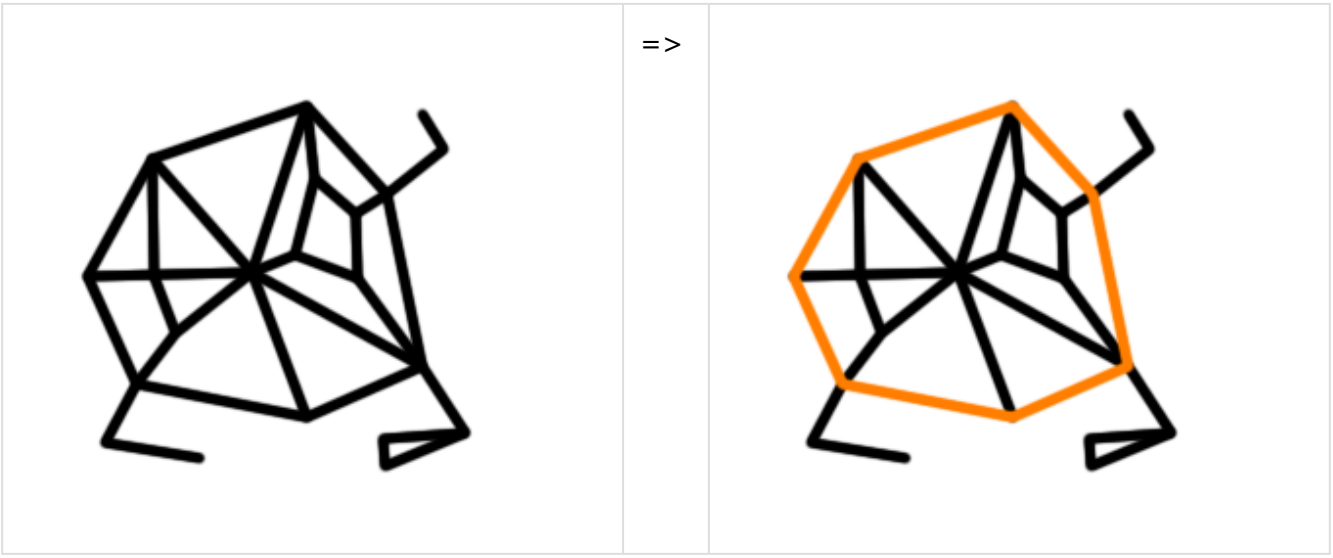
MINIMAL CYCLE BASIS

Introduction

In the Curve Algorithm we process through a number of Paths to find their intersections, then use these intersections and their neighbours to build the faces that find themselves inscribed in these intersections. In this page, we will visit the attempts to solve the last part of the algorithm, building the faces.

The first idea

It is possible to extract the larget Outerplanar Graph from a other graph as is shown in the following two images. We can find this Outerplanar Graph if we follow the steps that are next.

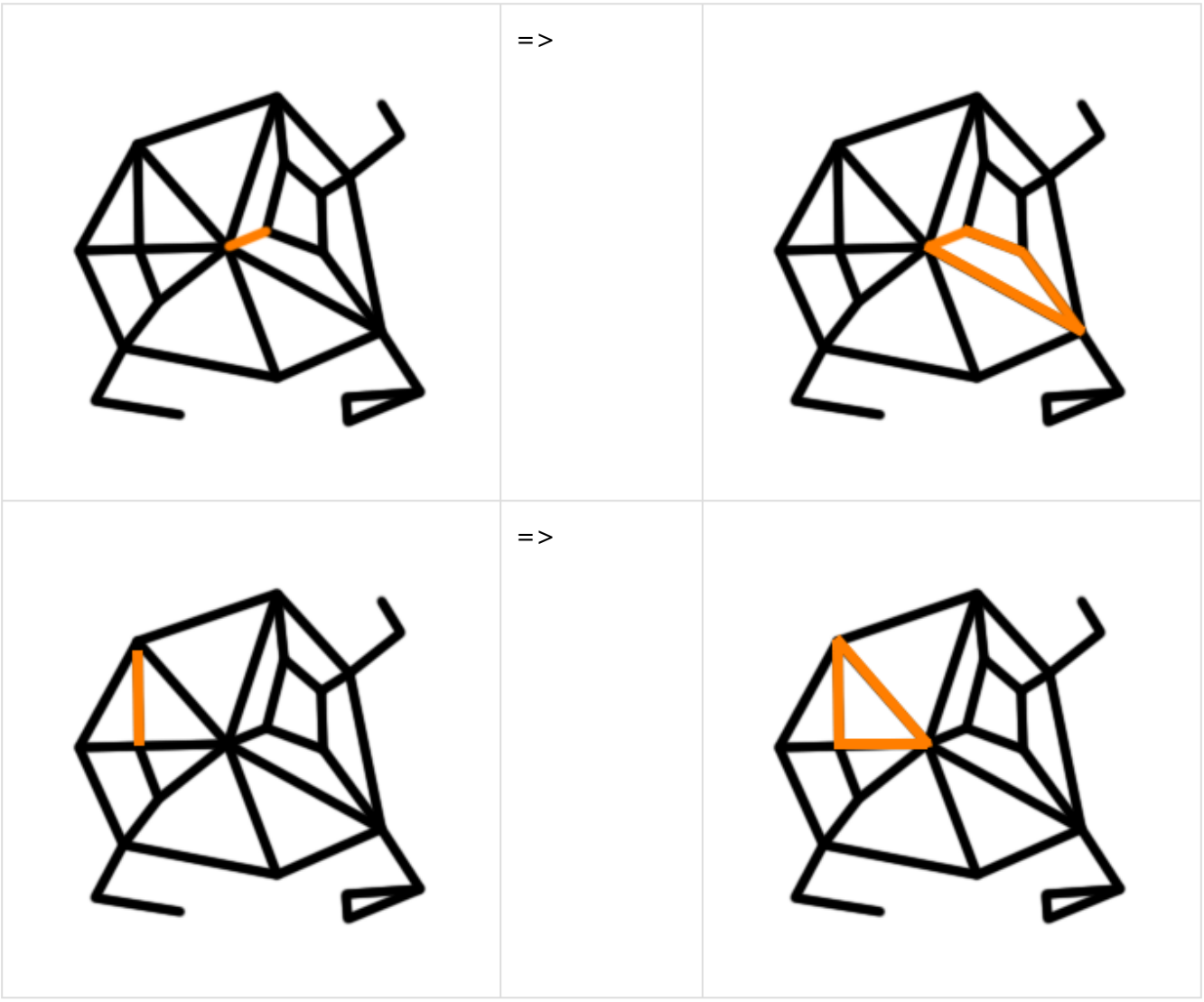


1. Choose a **Node** that is ensured to be outside any cycle. (So we'll choose the **Node** that is most to the left)
2. Choose the **Neighbour** of our **Node** that is the right-most (or the most clockwise) by basing ourselves on a point that is further than the chosen **Node** (as we chose the **Node** completely to the left we can choose a point a little more to the left)
3. We keep choosing neighbours that are the most to the right until we reach the starting Node.
    a. If we never reach the starting point, there are no cycles in this graph
    b. If we reach a dead end, we turn back until we find a **Node** that still has an undiscovered most right neighbour.
    c. If our result ends up with a list of X **Nodes** at the end that are identical-reversed of the X first Nodes, it means that our starting **Node** is in a dead end. So we simply remove those **Nodes**.

At least, that is what I have learned while working on a different project earlier in my career. I decided that this would be a good starting point for our problem.

It should be possible to modify this algorithm to find all the **Minimal Cycles** instead of an **Outerplanar Graph**. A quick example (shown below) shows that when you choose two neighbour

Nodes where at least one of the **Nodes** is inside the graph and you follow the previous algorithm you are ensured to find a **Minimal Cycle**.



Here are the problems that can be encountered with this simplistic approach :

1. If choose "wrongly" our two **Nodes** (like two **Nodes** that are outside the graph), we can find ourselves creating a cycle around the whole graph, juste like an **Outerplanar Graph**.
2. We can quite easily find more than once the same **Minimal Cycle**.

The second problem here could be fixed with an **Oriented Graph**, from which we would remove branches for each **Minimal Cycle** found.

Though after further discussion with my colleagues we could not find a solution to the first problem (not that there are no solutions, mostly that we couldn't think of one).

A*

In a later discussion, a certain colleague that gives the idea of using the A Star algorithm. The **A Star** algorithm will always find the shortest path between two points if there is one. So we could use this idea to find faces if we look for the shortest path between a **Node** and its **Neighbour** if we remove the edge between the two first.

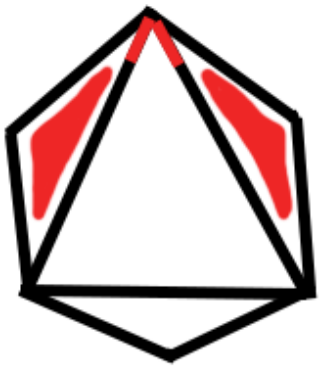
	
Le point rouge est le point de départ de la recherche et le point de l'autre bord de l'arrête orange est le point d'arrivée.	On enlève l'arrête entre les deux **Nodes.**
	
Avec A Star on trouve le chemin le plus court entre ces points.	Une fois un chemin trouvé on peut compléter la face.

Since we don't want to find the same face twice, we'll also remove the edge between the **Node** and the **Neighbour**, but only in one direction (**Node** → **Neighbour**).

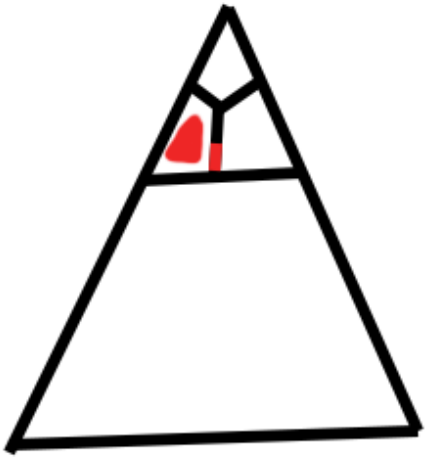
Next, we'll apply this algorithm for each **Neighbour** of the **Node**. Once we've gone through all the neighbours we keep going with the next **Node** until we've iterated through all **Nodes**. Sometimes we'll end up with no result as we won't be able to find the starting **Node**, but that is fine, we simply discard that search and keep searching.

Here are the problems that we currently have with the previous algorithm :

1. A face can be blocked and never found because of two edges on the face that are removed, but from different directions.



   a. (The red lines represent that the edge Not-Red to Red is blocked, and the red blobs are faces that have already been found)
   b. So in this case we can see that the center face can never be completed.
2. There are other edge cases where we will find the same face twice.



   a. (The red lines represent that the edge Not-Red to Red is blocked, and the red blobs are faces that have already been found)
   b. In this case, if we were to choose an edge on the bottom of the red face, we would only be blocked in a single direction by the completed edge (in red). So the face would be completed again, but in the opposite direction.

To solve the first point, we could force the selection of edges to always be cyclic. So instead of removing the edge Node→ Neighbour, we would the edge in a clockwise direction. We could find the clockwise direction by calculationg the curve orientation.

To solve the second point, we could remove the last edge of the face as well as the first. so if we have the face A→B →C, we would remove A→B and C→B. Though this would come against the previous problem and we would find ourselves in cases where faces are not accessible.

We must then keep searching to find which combination of edge removal can be made to solve our problems.

Minimal Cycle Basis

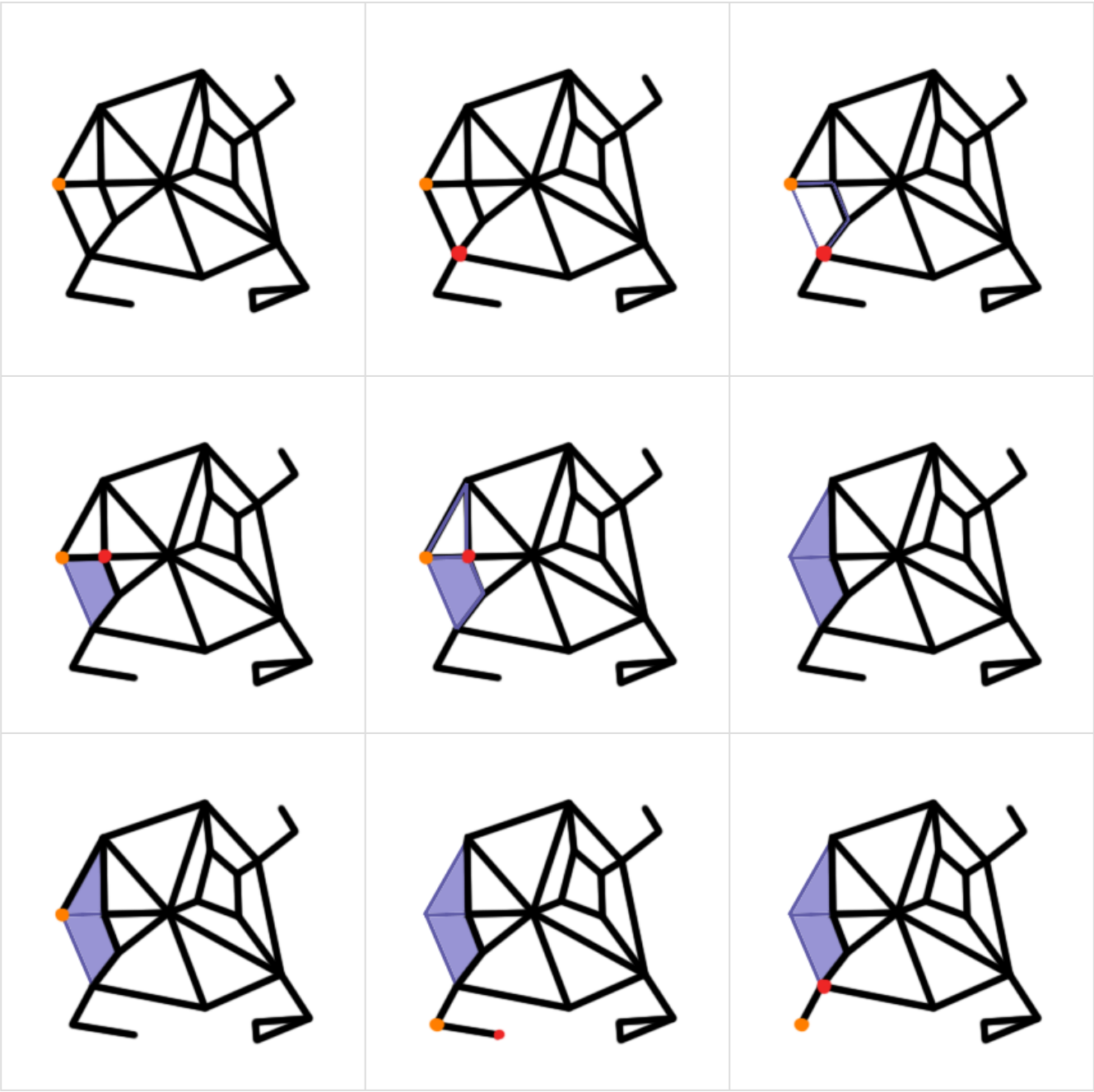
The same colleague that offered the **A Star** possiblity suddenly remembered of the existence of the **Minimal Cycle Basis** at about the same time as I was getting frustrated with the **A Star** algorithm.

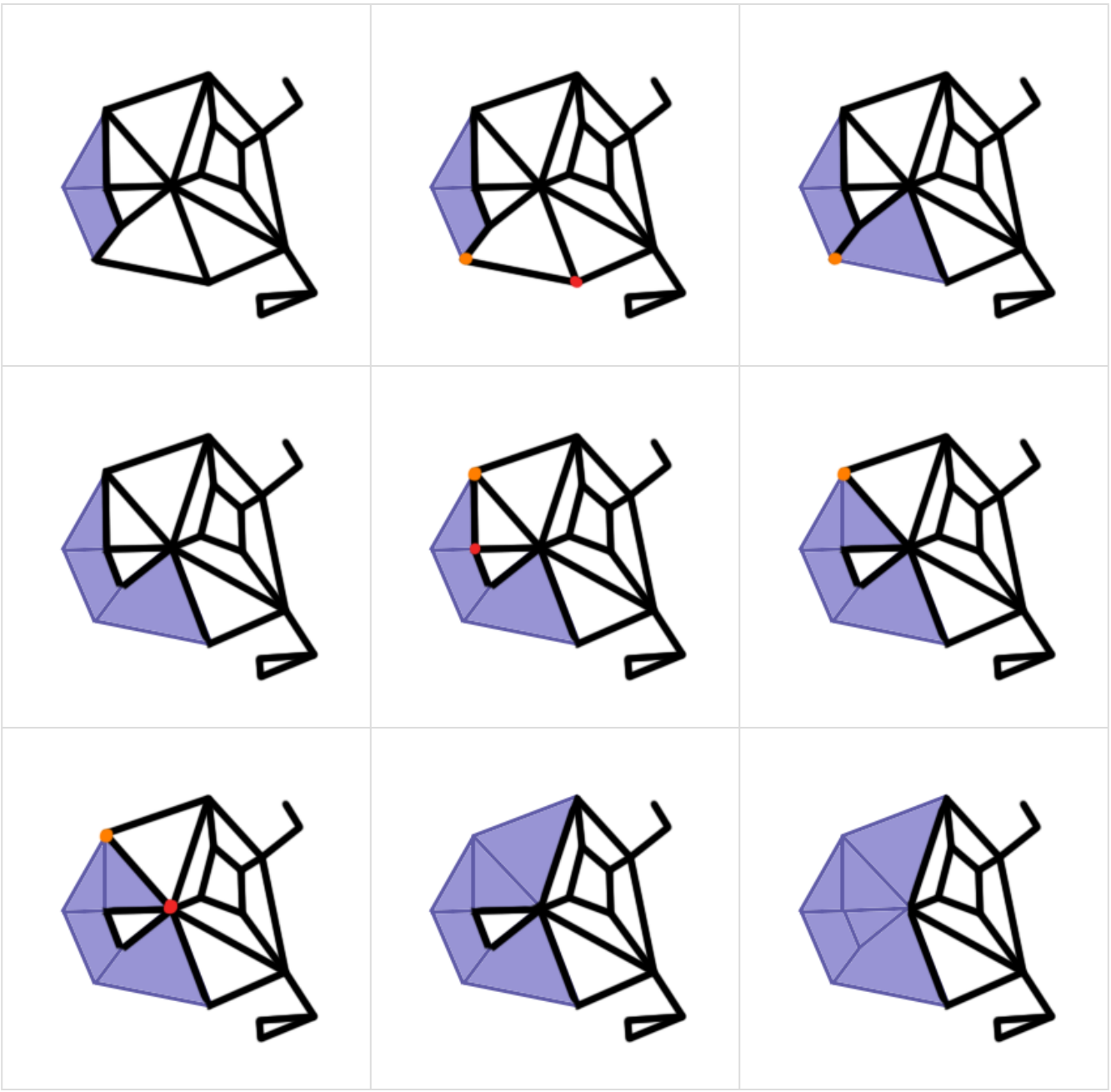
A **Minimal Cycle Basis** describes the problem of finding all **Minimal Cycles** in a graph, which is exactly the problem we are trying to solve. The **Minimal Cycle Basis** has already been solved and many solutions can be found. We will following the solution presented here : https://www.geometrictools.com/Documentation/MinimalCycleBasis.pdf.

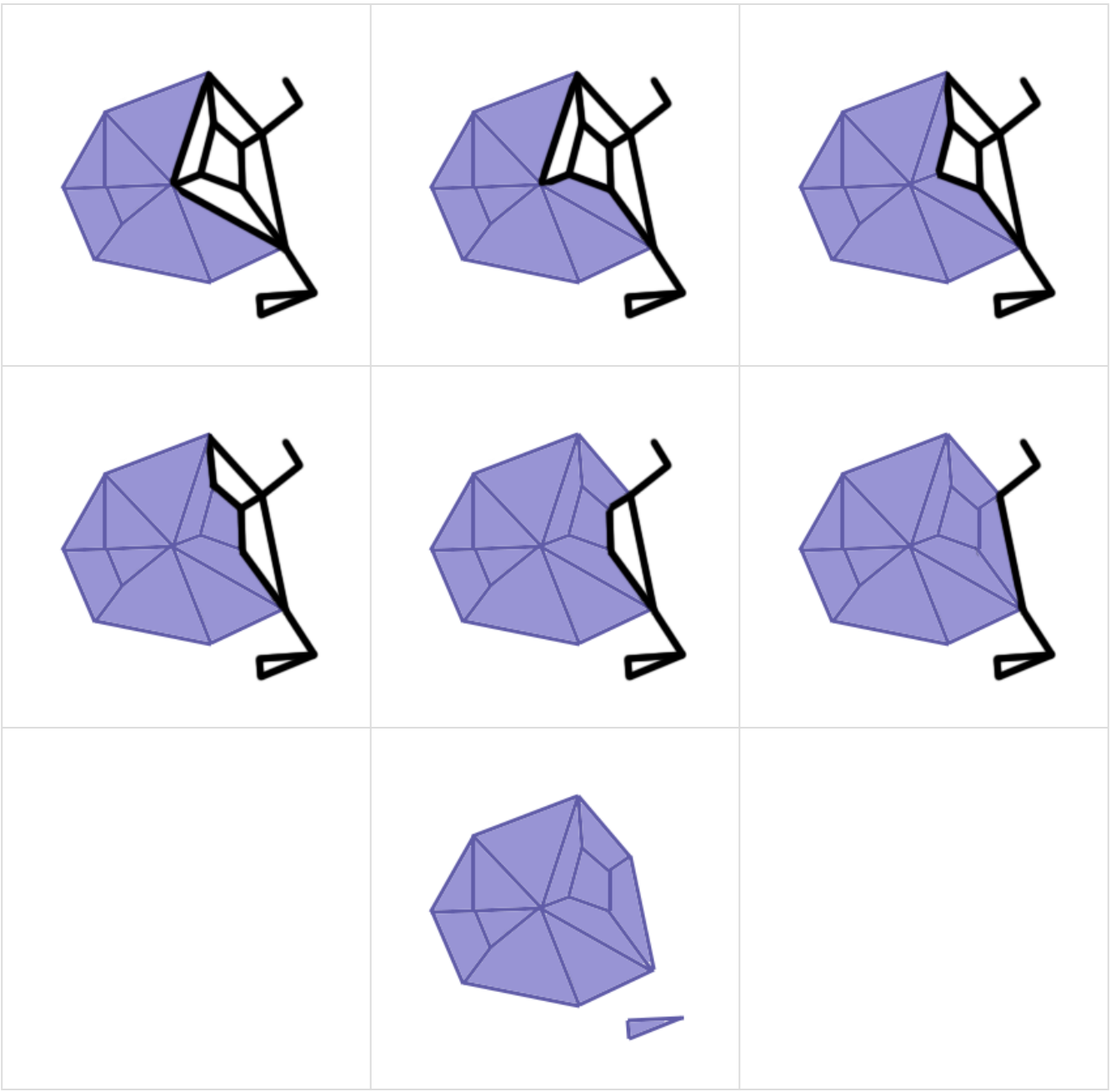
When I started reading the previous article, I realized immediately how close I was to the solution at the beginning of my searches. The idea to look to the right-most neighbour was a good, I only needed to turn left right after and keep turning left. Here is an explanation of the algorithm followed by a visual explanation.

1. Choose a **Node** that is ensured to be outside any cycle. (So we'll choose the **Node** that is most to the left)
2. Choose the **Neighbour** of our **Node** that is the right-most (or the most clockwise) by basing ourselves on a point that is further than the chosen **Node** (as we chose the **Node** completely to the left we can choose a point a little more to the left)
3. Next we choose a neighbour to **Neighbour** that is left-most and we keep looking for left-most neighbours untile we find our starting **Node**.
    a. If we can't find the starting **Node**, it means that we found ourselves in a dead-end. So we can simply abandon the **Neighbour**.
    b. It i spossible to find the starting **Node** but to still have a face that isn't valid. Ex. A **Node** is found more than once. In this case we can also abandon the face and the **Neighbour**.
4. After we find a valid face we remove the connection between the starting **Node** and the **Neighbour**.
5. We now return to the second step until the starting **Node** only has a single neighbour. At that point we return to the first step and find a new starting **Node**.

In this visual explanation, the orange dots are the starting **Nodes** and the red dots are the **Neighbours**. The purple faces are the newly added faces.







Its this algorithme that is currently implemented in the project. It is quite fast, especially compared to a previous step of the overarching algorithm, the step that iterates through all Bézier curves of each Path to find all intersections. The article on which this algorithm is based on does not offer an Asymptotic Annotation so we'll try to guess our own. It is my own short calculations that the algorithm should be of O(2N) which means that the time spent by the algorithm will only increase linearly when more intersections are added.