Contents

Foreword

Why Sketchfab

At the end of my 5 years at UTBM, I wanted to put a first step in the 3D area, for which I have a real interest. After having discovered it through my first internship at Voxelia (Strasbourg), I became to self-learn the basics of 3D through books, modeling with 3dsMax and trying to understand how all of this works. This is during this period that I heard about Sketchfab, this really interesting 3D start-up, mixing 3D engineering and web and having offices in both Paris and New-York. Sketchfab was my first target while I was looking for my final internship. I was also very interested by the environment the start-up was involved in. Working to make 3D artists able to share their awesome work, companies that want to share models of their new products or prototypes, and 3D printing which is in current democratization. After an interview and some mail exchanges, I finally had the opportunity to work for Sketchfab.

About the report

This report summarizes the 6 month spent at Sketchfab in Paris as 3D back-end engineer, from September 2014 to February 2015. I have discovered, learnt and deepened a lot of concepts around 3D rendering and 3D data processing that are required to explain the context of the different tasks I made at Sketchfab. Thereby, the report may have a more important part of materials as usual, to be more easily understandable and readable regardless of the technical knowing of the reader.

It begins with an presentation of the company, trough its history and its product, before introducing the internship and the main goals I set myself when starting. The following part will focus on the environment of work, the tools and the main workflow that are used at Sketchfab, before switching to the third part which explains the work done during the 6 months. I also took the freedom to make a dedicated part to mention the external activities I have participated in, before ending with an overview of the results, the observations and concluding.

Introduction

What is Sketchfab

History : from WebGL to Sketchfab

At the beginning of the second trimester of 2011, the web sphere made a step forward with the apparition of WebGL, allowing 3D rendering in the web browser without the need to set up any plugin. It was developed and released by the Khronos Group, consortium formed in 2000 and known for the famous cross-platform computer graphics API OpenGL. WebGL is a JS API for rendering 3D graphics into HTML5 canvas elements, and is based on OpenGL ES 2.0, an adaptation for Embedded Systems.

3D models are medias that are difficult to share : they require the installation of specific plugins, or in the worst case, the user needs to have the 3D software (used to create the models) installed in its computer. So far, 3D artists had to render their work in 2D into images files, or render a lot of frames and build a video sharable on Youtube or other broadcast websites. In both cases, it is difficult to see the model as a 3D media, including rotating around, zooming, and having real-time effects like reflections. Such off-line renderings have the advantage of giving very good results since there is no frame-rate constraint, but they have the drawback of being static.

> “We bring 3D to the web” [illustration : screenshot of the website]

With this need and the apparition of the native support of 3D in web browsers, 3D passionate Cédric Pinson released ShowwebGL in 2011, what we know now as Sketchfab. ShowwebGL has a very simple workflow : the artist just needs to upload a model using one of the supporter file formats, then he receives an url allowing him to share its work. At the beginning, Showebgl accepted a lot of 3D file formats (almost 20), and handled textures that were packed with the model into .ZIP archives.

Later, Cédric’s prototype gained a real interest for Alban Denoyel, sculptor and passionate by 3D printing, who joined him and put its business skills into the project. In March 2012, they create Sketchfab and are joined by Pierre-Antoine Passet (as CPO) six months later. At the end of the year, they integrate the Camping, a startup accelerator in Paris.

The 3D sharing platform

> “The Easiest Way to Share Your 3D Models” [illustration : screenshot of the website]

Today, Sketchfab counts around 150K users and around 250K models with a few ones which have exceeded 1 Million of views. The workflow remained pretty simple. After having created an account, the user just have to click on the "Upload" button, choose a 3D file and upload it. During the upload, he has the possibility to fill data fields if he wants to add informations about its model. He can provide a description, giving details about its workflow, the reason which motivate him to make the model, or what the model is made for. He can set a category for the model, and also add some tags to identify the content type and/or the tool used.

[Illustration right inlined : upload popup]

To improve the exchanges between users, a feature was recently developed to allow Sketchfab members to make their models downloadable, while defining the rights and their own price. With the democratization of 3D printing, it is very interesting for Sketchfab to be used as a 3D model database.

Basic account doesn't allow to upload files whose size exceed 50 Mo. To overpass this limit and also have access to more features, Sketchfab proposes two types of paid accounts. The PRO account moves the size limit to 200MB, allow the user to set its models as private. Private models are not shown in both the global feed and the user's page, they are only accessible through the URL. The user can also restrict the access to the model with a password. Some edition feature are also available, such as the use of annotations (pins with caption that are mapped to the model, up to 20 per model), the upload of custom background or basic viewer settings. Pro users can also use Sketchfab as their portfolio with a pre-build dedicated page. To answer the needs of companies that base their products on 3D models, such as architecture or prototyping, a BUSINESS account allows bigger models (up to 500 MB) and advanced customization. PRO and BUSINESS accounts offer support in 24h and 12h respectively. [illustration : include icons PRO and CORP in the text line]

In the community side, a forum was created to improve exchanges and conversations within Sketchfab members. It is a good users can ask for support or find the help they need, explain or get informations about workflows and techniques, and keep track of the Sketchfab's news and the events that are planned.

[Illustration : screen shot of the forum with fall-of at the bottom]

The editor After uploading the model, the user has access to a bunch of tools to customize and improve it's render through the Sketchfab built-in editor.

The first part is dedicated to the scene in its general aspects, with an interface to customize the camera, the background/environment, allowing the user to adapt the viewer to get the best result for its model. The very popular *camera effects* feature can be used to define post-processing effects, which are often considered as *Instagram like* filters affect the whole rendering.

The second tab contains the material settings. The user can tweak separately the scene material through a set of parameters. To be edited, the material can be selected trough the drop-down menu or by picking (double click) on one of the geometries in which it is applied to. This part is very useful since most of the time, the artist switches from the off-line render of its authoring tool to Sketchfab's real-time renderer. This implies a lot of changes due to the specificity of the 3D software (specific assets), and the performances constraints for real-time rendering. Moreover, Sketchfab offers built-in assets, such as backgrounds and environment textures (which are not so easy to use) to customize the environment around the model.

The third part is dedicated to the management of annotations. As said before, this feature allows the user to set a bunch of points (called hot spots) which are points on the models that the camera can snap to, with a text field to put comments or notes about the focused point. Annotations are very useful to give details about some specific parts of the model, their story, meaning or just to build a path to run step by step through the scene. [Illustration : sample of illustrated model]

After that, if the user is not happy about the render of its model, or if something is wrong with its 3D scene, he has the possibility to re-upload it, or delete it.

Browsing models Including a browser, the website has a lot of part dedicated to the community, and the models feed. Models that have a certain level of views and likes are pushed up to the "Popular" section, including the "Models of the week" and some related categories. The best models can also be "Staff picked" by the Sketchfab team, which mean they are at the top of the feed.

Who use Sketchfab ? Sketchfab is for anyone who want to share its 3D work. Architects or 3D artists from the video-game industry, design students or simple hobbyist, Sketchfab users are very diversified and come from all around the world. 3D printing companies use it to share their printable models, artists use it as portfolio to share their amazing work. Moreover, a lot of 3D authoring tools are natively plugged to Sketchfab through exporters or plugins, and several news or art websites include Sketchfab's embeds in the content of their web pages.

The Team

[Team organigramme]

Presentation of the internship

Context of work

At the beginning of the internship, the subject was not very precise but it concerned all kind of work that can be done from my job, as 3D back-end developer. For the main part, it involves all the operations that are applied to a given model right after its loading. At the global scope, the 3D back-end is about all the pipeline the inputs pass through before being stored in the database and being able to be shown on the website. It is composed by 3 main parts :

File reading and data parsing : The first stage the uploaded file passes through. It is globally managed by the OSG toolkit (detailed further in this report), and allow to both retrieve the file's data and generate metadata for the front-end part.

Data cleaning and normalization : since the platform is public oriented, we have to deal with files that come from a lot of sources, with a lot of file format that respect more or less the specifications. Through this second step, the retrieved data is cleaned (value clamping, various correction) to ensure that the data is consistent before applying the operations of the third step.

Model compression, preparation and optimizations : This last step has the function to prepare the data to be rendered using WebGL in the client side. Since WebGL is based on openGL ES, it inherits all the constraints that are imposed by the development for embedded systems. In addition, web browsers are also limited about the data they are able to manage. Finally, a such web platform needs to lower as possible the quantity of data to send to the clients. To reduce the loading times and the amount of memory used, and also improves the rendering performances, a set of operations including deduplication, compression and optimizations, are applied to the data.

So the main subject of my internship was to work on these different parts, with the help of the back-end engineer.

Prior Experience

What really helped me to get started working at Sketchfab is the ST00 I made the previous semester. During this personal project, I self-learned about the basis of 3D real-time rendering from both programmer books and 3D softwares. It was very important for me to have an big picture view of what is 3D actually is, because it allows to bypass the first feeling of complexness and abstractness about 3D. By working on both 3D engineering and artist sides, I had a useful feedback of how all of this works, and I can now better anticipate the result of what I do on the data. Practicing myself 3D modeling, I am also a Sketchfab user.

My goals as Sketchfab intern

I firstly choose to do this internship at Sketchfab to begin to work in a 3D based product. More than acquiring work experience and improving my programming skills, I aimed to learn the more basics as possible about 3D rendering and models processing. The other motivation I had was to get and understand the "big picture" view of a 3D web platform and all the community part it involves. From a more personal point of view, I was interested by learning more about 3D modeling and texturing, which is easier when having the opportunity to work and share with artists and hobbyists.

At the end of my studies, I am very interested by building my career as 3D software engineer, so I expected this to be my first start-up in the area.

Working at Sketchfab as 3D back-end engineer

The development environment

Every Sketchfab developer works on a virtual machine where all the required tools are built and run. Sketchfab uses Vagrant to host and managed virtual machines and a recipe software to set-up the environments. A local server is also run by the VM to be able to work on a local version of Sketchfab (very helpful for testing purpose).

At the beginning, I had the possibility to choose the operating system with which I wanted to work. According to the compatibility advantages and the productivity I can have thanks to my habits, I chose to work on Windows. As a student, I have access to a lot of free 3D software licenses from Autodesk (3DsMax, Maya) and also Unity. Under Windows, I was able to use these softwares in my work, and also use my knowing about them to help other developers that would potentially need some specific files or sample.

Since the VM has no graphical interface, I firstly worked using *vim*. A little bit later, I chose to move use *Sublime Text* (which is more user-friendly and have a lot of very helpful features) in parallel with a SFTP protocol to make the bridge between the host and the virtual machine.

The workflow

Project management and versioning

The project is principally managed using the Pivotal Tracker. Developed by Pivotal Labs, this software is an agile project management tool used to manage tasks, host files, track velocity, and plan iterations. Through tags and labels, it is easy to keep track of the work done, the work which is currently done, and the tasks that are in the queue.

Github, the well known web-based Git repository hosting service is used as versioning tool within Sketchfab. It provides a bunch of interesting tools for source code management and revision.

To improve the traceability of the work which is done on the project, Github and Pivotal Trackers are linked by the task IDs.

Workflow template

When beginning to work on a given subject, the corresponding task is started. The task is commented when needed, to discuss possibilities, decide which solution to use and everything that can be relevant. When a task is finished, the corresponding Github branch is used to create a Pull Request, waiting to be reviewed and commented by some members of the team. This step is primordial to spot potential bugs, discuss and propose improvements when needed. From a personal point of view, these Pull request were very a helpful source of learning. Comments were very useful to know what are the better ways to do things, what is not to do, and so on. I have learnt a lot of good practices thanks to these reviews.

Pull request validation For each pull request and after the code review, all the processing tests are run on a dedicated server to check if it can be merged without any problem. Jenkins, a continuous integration tool, has the responsibility of running the processing tests. When running a test on a OSG pull request, Jenkins checkouts both the OSG (work) and the ShowwebGL(tests) branches, updates its model base if new models were pushed into the model test repository. Then, it builds the tools (OSG) and set the environment before running all the processing tests. Tests are firstly run in local within the virtual machine to spot bugs. Jenkins test are then run to spot potential specific issues related to the environment or out-of-date plugin or tool. Jenkins is also useful for several tests that require a lot of ressources and that can't be tested using a local machine.

To the production Once the pull request is accepted, it is merged into the *develop* branch and the story is accepted. *develop* is used by every developer so it provides a first testing step. When a release is in preparation, the *develop* branch is built on a staging server which constitutes a second level of tests, with conditions that are closed to the production. After having tested on staging server, the code is pushed into production during the release. Everybody is asked to test the new features to spot potential bugs that were not spotted in the previous testing phases.

Communication inside Sketchfab

Internal IRC To improve the team communication and also reducing oral conversation which might disturb the other members of the co-working space, Sketchfab works with an internal IRC called Hipchat. It was very useful, especially to be aware of what is going on in the team, to share interesting articles, links or files, and ask for help if needed. It also permits to keep track of the conversations, and browse for missed or informations if needed. This point was very interesting and useful in my opinion, because it made me hear about concepts and techniques that are not directly related to my work, but are relevant for the general culture and the context of 3D development software. It was like a passive learning tool, and I really appreciated it during my time at Sketchfab.

Weekly overview and meetings Two meetings are planned on Monday. The first one is done in the morning and had the purpose of doing an overview of all the work that was done during the past week, ask question, highlight the encountered problems and discuss about what to do for the current week. They are the opportunity for each one to discuss, ask questions or propose solutions.

A team meeting was also planned each Monday afternoon, in video conference with the NYC business team, to talk about stats, the community work for the business side, and the features, the contents of the release and the incoming features for the technical side. I was also the moment to talk about the events, the feedbacks and all the other subjects that were done or improved at Sketchfab.

Feature Friday

During the last part of my internship was introduced the “Feature Friday” work sessions. The concept is to dedicate the last hours of the week on a project that mixes both personal and company’s interests. During this time, we are free to experiment and work on a subject we are interested in, make some tries or prototyping, if it respects some rules. The first constraint is temporal and asks for the feature Friday to be realizable in one or up to two sessions. The goal is not to spend a lot of time on it since the subject out of the company’s road map. The second rule is that the project must be “integrable” within Sketchfab, or at least represent an interest for it.

3D back end work with OSG

The beginning of the internship was a bit confusing for me since I had to understand and assimilate the Sketchfab’s pipeline in its overall. Being able to place my work and myself in this process was primary to do work efficiently.

Presentation of the pipeline

[Illustration pipeline with file screenshot and evolving small 3d model]

Right after the upload, the 3D file pass trough a pipeline before being stored in database and referenced to be showed on the website. This is actually a sequence of operations and processes whose goal is to read, correct if needed, and prepare the model to be registered and well rendered in the viewer.

Briefly, this pipeline is divided in three main parts :

1. **Data retrieval and cleaning** This part reads the file using the right plugin and writes them into a .osgb (osg binary). Textures are listed and data are cleaned to be consistent for further processing steps.
2. **Generic model building** The generic model contains the original data of the input file, rearranged in a more generic way. Its role is to have a homogeneous
3. **OSGJS file building** Here is handled the conversion from the generic model to a OSGJS model which is easily readable by the JavaScript side, in order to be rendered. A lot of operations are applied to optimize and compress the data to be rendered using WebGL.

Technologies and concepts

The whole pipeline is a set of scripts that are written in Bash and Python. These scripts are used to operate on the files, making them to pass through the adapted processed, and to manage the use of the different tools.

As base tool to manage 3D data, Sketchfab uses the OpenSceneGraph API. Open-source and cross-platform, this toolkit is written in C++ and distributed under a LGPL like free license. Based on OpenGL, OpenSceneGraph is a rendering-only tool, it doesn't provide higher functionality to be used for gaming purpose for example. OSG renders a 3D scene using a scene graph structure and offers a lot of tool to manage and optimize the data.

Some words about scene graphs A scene graph is a data structure representing a 3D scene. It contains all the spatial and logical relationships between the scene object, leading to an efficient management and rendering of graphics data. The scene graph can be seen as a hierarchical graph composed of nodes, with a top root node linked to child nodes. In a scene graph, the transformation applied to a node also affects its children. This structure make it easier to transforms objects and create animations.0

Place and responsibilities

As back-end engineer, my work concerns the whole pipeline but I mostly worked on the reading and cleaning parts. Working with OSG plugins, I had to ensure that the input file was correctly read and processed without missing or misreading data. The variance of the input is high at this step, so I had to manage a lot of edge cases that can make the processing to fail. We don't have any control of the data that are actually sent by the users, but the main goal is to correctly return the model if the file is consistent. If it is not, the goal is to be able to identify clearly what is wrong with the file and return an accurate error message to the user.

OSG plugins are used to read the input 3D files coming from the Sketchfab users. With the apparition and the development of 3D editing tools, some new features and data are added to these 3D file formats, in a specific or a more general way. The majority of the OSG plugins were written to read 3D files in a basic way, without handling all the data the file can contain. In the best cases, data are missed and the model didn't shows as expected in the viewer, and in the worst ones, the processing failed because of inconsistent or mishandled data in the files. Moreover, the front-end may need some specific meta data that are not useful for 3D rendering (so no parsed by the plugin) but are relevant in some cases to improve the model settings or the user experience.

Retrieving models from Sketchfab

In addition to the issued models returned by the support, Sketchfab has some mechanisms to retrieve issued models from the production database. If a model is bad rendered, we can ask for it to be rsynced (copied from a server to another) copied on a server from which we can retrieve it. This system allows to identify the problem the user had, and to have useful resources for testing purpose. Moreover, a dedicated channel was set in Hipchat to monitor the activity on Github, Pivotal but also on production servers. When a model fails to process, an error message is shown in the channel with the Id of the model and the command to retrieve it.

I managed these different sources to work efficiently and test my code. I also used to exchange with the community support to help in identifying more accurately the problems and explaining how to fix it, when is was possible.

Public and specific code

Sketchfab uses contribute on OSG by reporting the changes made on the forked OSG repository. Including the changes into the main repository has the advantage : it makes it easier to update it by reducing the deltas between both versions.

On the other hand, a part of the added code and behaviors is very specific to Sketchfab and needs to be distinguished and kept away from the submitted one.

This *dual* workflow adds a level of difficulty since these two parts need to remain separated. To facilitate the submission, the specific code has to be aggregated, that requires the developer to adapt the structure and also the logic of the new code. Moreover, the weight of the public modifications needs to be as minimized as possible to be more easily accepted and then merged into the main code. In some cases, it can require additional work and time in order to lesser the impact of the modification while keeping the code clear, optimize and logical.

Fixes and Improvements

Working on plug-in issues was a little different of working on a project. It requires a lot of reflexion and understanding of the existing code to find a solution. During these different works, I spend the most of the time reading the code and trying to understand the whole process before being able to spot where the issue were located and looking for a solution. The solutions are finally not very complicated, but the difficult part is to find where to fix it, taking into account the whole process and avoiding all the side effect the modifications can cause. I personally found this very interesting for learning 3D concepts.

Moreover, in some cases I had to begin by refactoring the existing code before adding the new one. Working on a shared project require the developers to care more about the readability, the quality and the maintainability of the code. In this case, it has to be easily understandable to allow OSG users to use or improve it if needed. These are some rules that are primary in software engineering, whatever the concerned project.

Normals consistency and correctness

Normals correctness and consistency was the first topic I work at Sketchfab. A lot of uploaded models had rendering issues that were generally related to these data, so Sketchfab needed to have some code to solve this.

Normals in geometry data

A 3D model is composed of vertexes that define faces. Basically, a vertex is defined by its position (is local space), with a set of data such as normals, vertex colors or texture coordinates. Vertex data can be mapped per faces, but this mapping is more generally done per vertex. [Illustration : model wireframe -> vertex -> vertex data]

As the name suggests, these data are used to define the normal of a surface at a given point : they are always represented as 3D vectors. At the render step, normal are mostly taken into account for lighting et reflection calculation. They can also be used to compute effects such as refraction or in more complex effects such ad BRDF (for Bidirectionnal reflectance diffusion functions) that varies according to the surface properties.

In the case of **flat shading**, all the vertexes of a given face have the same normal and the face is lit uniformly in all its surface. For **smooth shading**, these vertices may have different normal values, that are applied to the whole surface by linear interpolation. This generates a continuous and homogeneous lighting of the surface and allows to create smooth surfaces without the need of a lot of vertexes.

[Illustration : Flat vs smooth]

Deviation tolerance

Faces are considered co-planar, which means that all of its vertices are contained in the same plane. The normals of each can be oriented in whatever direction, but needs to be included in the hemisphere oriented towards the implicit face normal.

In the reality, there are some cases where the data doesn't obey this rule, leading to unnatural and weird lighting on the corresponding surface.

[Illustration : bad normal orientation]

I first focus on this case and wrote some code to visit all the geometries of the scene and check the normal orientation. The goal was to be able to tell the user that its models has normals that don't respect this rule and may not render correctly. A bad normal data can have a huge impact on the model, being responsible of rendering issues such as black parts or weird face appearance. For this purpose, we cannot automatically recompute the bad values for the users, since this misorientation can be willingly set to achieve some special effects.

So the bad normal detection consists of checking the deviation of each vertex normal with the face's one and ensure that it is lower than 90°. It it deviates too far, a warning is output.

[Illustration : arc with normal and illumination rendering]

[Illustration : bad normals showed and problem targeted]

Normal consistency

To go further in the problem, some models actually have normal data that are inconsistent and leads to more important rendering issues.

Normals are mostly bound PER_VERTEX (per face is deprecated), that means that we need a one to one mapping between the vertexes and normals arrays.

Some models don't have any normal data, others have some but not enough, which leads to vertexes without normal data, and some other have nulls normals. Exporters don't always write normal data, it is often optional and values are recomputed during the reading step.

Basically, the lighting of a surface is computed using the dot product between the light vector and the normalized normal of the surface, so a null normal leads to an unlit (or partially unlit) surface.

OSG provides a lot of tools to deal with geometries and the scene graph in general, based on the design pattern *Visitor* and *Functor*. The visitor is in this case very useful and powerful since it allows to apply a function in a set of similar objects.

To smooth a geometry, a visitor will visit the geometry's normals and recompute them according to its topology.

We differentiate face normals and vertex normals. Each face of a geometry is supposed to have its vertexes coplanar. In most of the cases, primitives are triangles, which ensure that this rule is respected, and only quads are supposed to be planar. As a consequence, a face has a unique normal which is the normal of the plane it belongs to. Another way to define the normal of a face is by interpolating the normal values of its vertexes. This allows to obtain what is called *smooth shading*, which avoids the model to appear faceted by giving the illusion of a smooth surface. For each triangle :

1. Compute the faces' normal and add the value to the three vertices's normal
2. Normalize the normal
3. Check for normals that deviate too far, according to a crease angle value The crease angle is used to determine the maximum angle allowed between two adjacent faces.
4. If the normals of two adjacent triangles deviate too far, the shared vertexes are duplicated. [Illustration : triangles duplicated]
5. Recomputes the normals for the new topology

In order to be run on all models, this code was put in a cleaner pseudo loader which is called after the 3D plug-ins. Pseudo loaders are plug-ins that can be plugged right after any reader to perform post operations on the model or more generally on the data that was previously parsed.

Normals were the first data I dealt with, and were a good point to begin learning to understand and manage 3D data.

Warning messages through the processing

The cleaner's purpose is firstly to clean the data that is not consistent, but also to do some checks and inform the user that the data he submitted has some issues. When detecting a bad normal or performing a data correction, a warning message is written into the standard output and is then flushed into a log file.

The problem of a such output is that it can be very tedious to parse as it contains every single processing output. To simplify the process, a python script has the responsibility of parsing the processed.log file and report the messages into a JSON file which is more readable by JavaScript code. To do so, I added functions in the existing processing python code code to allow to store warnings according to the type of data they refer to. It organizes the messages within categories which will be useful for the front-end part to return get and return the messages to the user

First, each warning message had to obey to a certain syntax to be retrieved by the meta logger script : `Warning: [class::method] [[type]] message` This syntax is based on the common form of OSG log messages, containing both the class and the method's names from which it is emitted, but also including a structure adapted for our purpose : categorizing the type of data the warning is about.

For now, these logs are used to keep track of normal problems encountered in the uploaded models. A feature is currently in development and will add an intermediate *draft* step between the moment he file is processed and the moment it is visible on the website. This intermediate step will be useful for the user to spot eventual issues and have a degree of freedom before having its model published. The processing warnings will be output to the user at this step, allowing him to reprocess or correct the model himself.

The following code is responsible of parsing the processing log file, retrieving the warning message and logging them into the JSON file.

```
function parse_processing_log() {
    Expects the log format : "Warning: [class::method] [[label]] message"
    Removes the "Level: [class::method]" before calling metalogger.py

    if [ -e "${LOG}" ]
    then
        grep '^Warning:' "${LOG}" | sed 's/Warning:\s*//' | sed 's/^\[[a-zA-Z0-9 :()]*\]//'
            ./metalogger.py "${metajson}" log_warning "${line}"
        done

        grep '^Anomaly:' "${LOG}" | sed 's/Anomaly:\s*//' | sed 's/^\[[a-zA-Z0-9 :()]*\]//'
            ./metalogger.py "${metajson}" log_anomaly "${line}"
        done
    fi
}
```

[Screenshot : JSON part with corresponding warning messages]

This cleaning code now used in production, but a bunch of models were still rendered with black faces issues. After this small project, I focused more on issues that are related to the plugins themselves.

Plug-in fixes

Plugins are not the main part of OSG, the majority of them was developed to enhance the 3D files support. Actual plugins were developed for personal use by the OSG community member and were submitted a little bit later to be integrated in the main repository. As a consequence, code is very different from a plugin to another and the data they manage are very closed to the specific use the developed needed on the moment of the development. Thus, they are not close to the specification of the format they are used to read, and may have some buggy or week parts that can easily fail or break.

With the huge variance in the sources and in the data contained in the files that Sketchfab receives, these plugins need to be improved to handle the more edge cases as possible. This what I aimed to do for the following tasks of this part.

Bad vertex colors for STL files

I firstly focused on the plugin used to read STL files. A part of the models it output had black faces issues. The common approach for this type of issue is to first check the format specification to understand the structure of the files and the data that is contained. Then, the osg has to be check to see if it fits the specification of the supported format. This allows to dissociate the cases of badly exported 3D files with the case of a buggy plugin code which needs to be fixed and improved. As said previously, the variance is high at this step and a lot of uploaded files don't have consistent data or are not conform to the specifications. In this last case, the plugin doesn't have to handle it, and this is out of the responsibility of Sketchfab.

Looking for the file specification STL (for STereoLithography) is a file format developed by 3D Systems for their stereolithography CAD software. Now, this 3D file format became a standard widely used for rapid prototyping and computed-aided manufacturing. STL is also widely used in 3D print because of its simple structure and the way it declares geometries.The STL format specifies both ASCII and binary representation, but the second is more common since it is more compact.

Within the ASCII representation, 3D Geometries data is given between `solid/endsolid` bounds, with a sequence of triangular facets. For each facet, a facet normal is given, followed by the three positions of the vertexes that define the faces. On of the major drawback of this format for rendering purpose is that normals are given per faces (and not per vertexes) which is not in adequacy with the per-vertex mapping in OSG : STL are faceted.

[Illustration : screenshot of a simple STL file]

STL doesn't support any other feature such as materials, transforms or relationship between objects. However, contrary to ASCII files, binary ones can handle vertex colors. Given for each facets (such as normals), colors may have different meaning according to the 3D tool they come from. For common softwares, STL colors are encoded on 16 bits, with 5 bits for blue, green and red, using the last bit to indicate if the color is valid (i.e must be used) or not.

Nonetheless, Binary STL files' structure may. differs from a software to another. For *iMaterialize 's Magic* software, the last 16 bits are differently used : they code the RGB color on 15 bits (so reversed compared to the common case) and the last one has another signification. The purpose of the last bit in the color field is to indicate if the given facet uses its own color or the main one which is declared in the header.

Finding the issue Looking at the model in shadeless rendering (without lighting), parts were still black. After having asked about the shader equations, It seemed to be related to bas vertex colors, and not the normals as I thought first. Moreover, all these files were binary encoded STL.

[Code : shader equation with vertex colors component]

Normals are used within the shader to render face's illumination, but in the case of a shadeless render, lighting is not taken into account. According to the shader equation, bad vertexes colors (blacks for the worse case) have a very high weight on the final color.

The fix consisted on first, generating the right vertex colors according to the type of software the file came from, and setting these colors per vertexes and not per faces are it was originally made. Binding vertexes data per face (said per "primitives") is more and more deprecated in favor of per vertex binding.

[Illustration : corrected STL]

Fixing this kind of plugin issues was a good introduction to understand the process of retrieving 3D data from file and formating it in order to be output through an OSG object. Data retrieving issues were pretty easy to fix when the whole process is understood. I managed to work on higher level issues that were related to the topologies. By topology, we define the layout of primitives that compose a 3D geometry. Working on OSG plug-ins led me to deal with higher level, not related to vertex data but concerning the geometry itself.

Geometry issues

Primitive sets Basically, the render of an geometry is called by sending a set of vertex buffers to the GPU to be rendered. Buffers can be of several types according to the type of data they contain : normal vectors and position that are 3D vector, or vertex colors that are represented with 4D vectors (for R, G, B, A). Buffers can also contain texture coordinates (2D vectors) or vertex attributes, that can be used to store tangents, binormal vectors, or any relevant data for rendering the geometry.

There are actually two types of structures to send this data. Draw arrays consists of sending buffers that contains all the values for each face. To reduce the size of these vertex buffers, OpenGL allows to use indexation. If a set of faces share vertex, their are not duplicate as in draw arrays, but the vertex data is unique in the buffers and each face will reference it through its indexes. A primitive set is composed of one of these two structures, the type and the number of primitive to render. The following picture shows the primitives types that are supported by WebGL : [Illustration primitive types : webGL_primitives from http://math.hws.edu/eck/cs424/s12/notes/jan20.html] POINTS, LINES, LINE_LOOP, LINE_STROP, TRIANGLES, TRIANGLES_STRIP, TRIANGLE_FAN

For each tuple of data, the GPU will draw the corresponding face, but introducing an offset in the indexes has huge consequences on the result. I also had to fix this kind of issues with OSG plugins.

A topology issue was reported by a user which used to upload furnitures using the 3DS file format. The given model had some part that were broken.

[Illustration : broken model]

3DS plug-in fix [Illustration : 3ds icon] 3DS if one of the file formats which is used to export 3D scenes from Autodesk's 3dsMax software. It was the native file format of the old Autodesk 3D Studio DOS. For legacy reasons, 3Ds is very limited on the data it can contain. Normals are not written, smoothing groups are used instead to help the software to generated them as good as possible. Only triangles are supported (so plug-ins may handle tessellation), and a mesh can't have more than 65536 vertexes or polygons.

Importing the file in 3dsMax or Blender was fine, but this doesn't give much more informations since the importer can modify the data between the moment of it reads the file and the moment it loads it in the software. Doing this has sens since warnings messages can be output by the software and give some details about the potentially corrupted data.

As said previously, in OSG the loaded object needs to have its normals declared per vertex, but the 3Ds specification doesn't handle this structure. The plug-in needs to generated them by its own. The issue appeared to be a side effect of this process. When a shared vertex needs to have two different normal values, the plug-in just split it and remaps the faces indexes accordingly. The problem is that the code was written to fit the 3DS specification which doesn't allow more than 65536 vertexes or faces per mesh, so data types are used in consequence : index are short int. Introducing a duplication in this case is dangerous since it will potentially increase the number of vertexes, thus the indexes, and finally lead to an overflow. This bug was not really obvious on the first look, I first thought that it was the result of a tessellation operated on a bad geometry.

In this case, duplication made sens since normal has to be consistent within the whole geometry. A vertex cannot have its normal deviating too far from the normal of the adjacent faces. So the fix had to be done on the types, by changing shorts to int. Primitive sets also needed to be modified. OSG differentiates three types of draw elements objects, according to the indexes data types : *DrawElementsUByte*, *DrawElementsUShort* and *DrawElementUInt*.

Result : [Illustration : good result]

Issue with shared arrays Even after having solved the 3DS plugin, these geometry issues were still found with some models. This looked to be an isolated case since I only had one sample to work on. Opening it in the 3D software was fine, but there was something in the pipeline that cause the geometry to be broken. The only thing that showed up was that the model had bad normals and was smoothed by the cleaner during the processing.

To determine which stage causes the issues, we use to disabled step by step the processing operations. Most of them affects the geometry, in its vertex data or its topology, so I looked that one of the step was responsible of the rendering issue. Doing this, it appeared that disabling the cleaner solved the issue, but It was not obvious since it only has to deal with the normal data, not the topology. The only part that was not fully *under control* was the smoothing visitor step, so I took a look into the code.

The geometry had in fact shared buffers for texture coordinates. Texture coordinates art part of vertex data and are used to determine the way a texture is mapped onto a surface. Since OSG doesn't support multi texture materials, when several textures are mapped onto a geometry, its vertexes need to have a texture coordinates array for each, even if the values are the same. To avoid unnecessary duplications, both buffers share the same set of data.

Looking back to the behavior of the smoothing visitor, it duplicates vertexes if their adjacent faces have normals that deviate too far from each other. During the duplication step, vertex data are duplicated and indexes are updated, but it appeared that the existing duplication code didn't take into account the case of shared arrays. Managing two times the same set of data (for each texture coordinates buffer) introduced an offset when the remapping was done on the new vertex buffer. The solution was to deduplicate temporally the shared data before the vertex duplication and restore the state right after the process. It led to a very useful fix that was merged into the main OSG repository.

In this step, I had a great example of the issues side effects can cause.

The next part of the work done on the plugins had the purpose of improving the supported features and more generally improving the way plugins retrieve the data from 3D files. I essentially worked on the FBX file format, which is one of the most powerful and complete. Sketchfab aims to use it as its standard file format since it is very rich.

Other fixes

The fixes I previously detailed were not the only I worked on. I also worked on small fixes and improved existing code in the plugins when It was needed or when it was worth enough. These works were about refactoring or changing the way operations were made (such as the normal retrieval or computation within each plugins), or simply cleaning the plugins of unwanted or buggy behaviors. For example, I improved the way options were retrieved within the STL plugin, and added an option to avoid the OBJ plugin to reverse faces when the normal was not consistent (that led to rendering issues to).

Plugins refinements

Options and refactoring When adding new behaviors to an existing code, It is important to keep it consistent and keep its logic. For huge changes, like FBX, It was very important to maintain the logic of the whole plugin. In a lot of cases, I had to refactor a part of the code before adding my changes. In software engineering, we need to follow some guidelines to make an efficient refactoring. With the help of my colleagues and good references on the web, I learnt how to refactor to obtain a code that makes sens. I also understood all the interest of the regression tests in this work : they are basically required to prevent from breaking the existing behaviors.

Avoiding to break behaviors in also a problematic when working on code snippets that are used by a lot of developers. For this, I managed to put the new behaviors as optional, in order to prevent any issue with the existing code based on OSG when needed.

FBX

[Illustration(inline) : FBX logo + autodesk logo]

FBX is a very powerful 3D format own by Autodesk, which is the most influent 3D softwares editor. It owns all the most used and powerful 3D authoring tools, such as 3DsMAx, Maya, Mudbox, but also AutoCAD. FBX (for FilmBox) was designed to serve as a bridge format between these softwares, and is very complete. Contrary to STL or OBJ, FBX contains rich sets of data about the whole scene graph and the different 3D assets. FBX was originally developed by Kaydara for its software Filmbox. Its purpose was to record animation data from motion capture devices. FBX uses a object-based model, allowing the storing animations along with 2D, 3D, audio and video data. Textures can also be embedded within the fbx file.

FBX is a proprietary format, but a SDK is also provided by Autodesk to increase its use. FBX exists in both ASCII and binary formats, and the standards are updated yearly (a given FBX is written according to a given version). OSG provides a FBX plugin based on the C++ SDK but the file format is not fully parsed. As the other plugins, this one was developed to fit with some users' specific needs. At this moment, the plugin suffered of a big lack of supported features, but was widely used by Sketchfab users.

I improved the plugin on both the way it manages topologies and the way it supports materials and textures maps. I began to work on the primitives support. In fact, FBX was able to parse any type of topology but the output was always composed of triangles.

Geometries and wireframe The GPU is only capable of dealing with triangles, so each sent geometry is internally tessellated (triangulates) before being sent to the GPU. When rendered or managed by softwares, models can be composed of more types of primitives : it can also have quads or polygons, which have some advantages for building and editing purposes.

The wireframe is a rendering mode which only renders the lines that compose these primitives. Artists care a lot with it, because it is a way to show the quality of the work done.

The support reported some mails from users that asked why the wireframe of their models were still broken into triangles instead of being the same as the one they had in their authoring tool, so I managed to improve this in the FBX plugin.

I worked on the FBX plugin to make it support the other primitive types to avoid modifying the original topology of the model. Looking at the FBX format specifications, all primitives are supported, but taking a look into the plugin code showed that everything was automatically tessellated into triangles. This is not a design problem or a bad choice, the developer just didn't care of the topology when writing the plugin. For our purpose, we had to handle the quads and the polygons in order to keep the wireframe unchanged. To keep it modular and prevent the existing code to break, I made the new behavior to be executed according to a *keepPrimitive* option. By default, the behavior remains the same. This option is also added in the processing variables to be enabled for our own use at Sketchfab.

[Illustration : difference with wireframe . Create a custom piece for this]

One of the difficulty of this part was to add the new behavior without breaking the logic and keeping a readable code. As said previously, changes have to be done in a such way the submission has to be easily accepted. This feature was made to be pushed in the official OSG repository.

After that, I focused on the way materials and textures were retrieved from FBX files. We had several reports from users about textures that were lost when uploading on Sketchfab, causing the model to appear very differently.

Maps and Materials Map is a term to define textures of different types that have an influence on the material they are assigned to. The term refers to the fact that values are *mapped* onto an image through the RGBA values. Maps can be used to affect floats (factors) and vectors (colors,normals).

Textures are mapped on geometry according UV coordinates that are generally represented as 2D vectors. For a given triangle, vertex are mapped onto the texture according to their own coordinates, and the rest of the face's points are mapped by interpolation between these values. [illustration : triangle, texture, mapping]

In 3D, a material determines the way an object reacts with the lighting and its appearance. When assigned to a geometry, a material is used to defines the colors of the object's surface. In most of the cases, the material has several textures that are mapped onto the geometry and mixed to produce the given result. OSG doesn't support multi-material, so the geometry has to have a texture coordinates (also called UV) channel for each used map.

More precisely, a material is composed of a set of parameters and maps whose combination is defined within a shader. A shader is globally a small program that takes 3D data as input and outputs a pixel color. So the appearance of a material and its behavior with the lighting depends on both the input data and the way the shader manage them.

[Illustration : material main schema with shader and result]

Parameters can be colors, factors, booleans depending on the type of values they encode. Object are generally textures, which can be considered as an object composed by an image and a set of parameters defining how to utilize it.

A material is divided in several channels, having their own parameters and interpretation within the shader. Even if each software may have its own channel naming and definition, it is always possible to find relations between them. The channels that are going to be detailed are relative to real time rendering. Some channels are reserved for off-line rendering since they require a lot of calculations and are very difficult to use in real-time.

Here is the list of channels that are used in Sketchfab and their function:

[Illustration : Material editor screenshot at the left]

- Diffuse: The diffuse channel defines the main color of the material. Even if it can basically be a simple color, it often uses a texture. The lighting shades the color according to the model's surface and the light(s) position(s).

- Specular: The specular is used to simulate the reflection of the light by a shiny surface. Here, the channel has two parameters : the color of the specular spot and the glossiness, which can be identified as the "radius" of the spot. Glossiness is also called shininess since it defines the shininess of the surface.

- Normal/Bump Map: This channel is used to add details on the surface by altering its lighting. Given a flat surface, the shader will use the set texture to perturb the lighting on the surface, altering its normals, which allows to fake bumps and dents. This channel uses two types of texture : the bump texture, which is a heightmap, and a normal texture which defines the surface's normal (x, y, z) by translating them into the (r, g, b) space. Normal mapping is detailed further in the tangent space part.

- Lightmap: As its name indicate, the lightmap is used to define the lighting of a surface using a texture. It is commonly used to add shadows to the model without having to generate them by calculation. This process is called texture "baking".

- Emission: The emission is used to make a surface glow, or making it insensitive to the lighting. It is commonly used for object that release light (even if the light is not often casted into surrounding objects)

- Environmental Reflection : Defines what is reflected by the surface. They are several ways to generate reflections such as ray-tracing (which consists of throwing rays and getting the point they are reflected to), but they are very expensive to compute to be efficient in real-time rendering. In the case of real-time rendering, reflection is made by pre-computing a special texture (cube map) and mapping it to the object. Reflections and specular behavior are linked here. The level of reflectiveness varies according to the specular level.

- Face Rendering: This is a material parameters defining if the shader has to render only the front face, or both the front and back faces. If front-face only is set and the face's normal points in the same direction as the camera, the face will not be rendered.

[Illustration : make a sphere with incremental rendering while adding maps]

At the beginning, only half of these map were supported by the FBX plugin, other were just skipped(lost). With the help of the FBX SDK documentation, I added the code to retrieve the missing maps and manage their related data in order to have them in Sketchfab. The corresponding texture coordinates also needed to be retrieved from the geometries to prevent mapping issues in the viewer.

At this step, map were retrieved correctly but the user had to set all of them using the editor.

Setting metadata to keep track of maps Maps were here, but the link between them and the channels was lost so the front end part didn't had enough data to automatically re set the maps into their respective channels.

Looking to the internal structure of OpenGL (and so OSG), there are no existing semantic to determine how to map the textures onto channels. The concept of channel is abstract : it is actually a texture unit.

In openGL, a texture unit is a texture object which packs together an image (texture) and a set of sampling and texture parameters. They are identified and linked to the OpenGL context using indexes. The order they are set is arbitrary, so it varies depending of the implementation.

In our case, connections are lost since Sketchfab interprets differently the texture units than the FBX plugin does. OSG has some containers which allows to store custom data on objects. These containers are very helpful to keep track of the texture units' mapping through the processing.

The *User values* consist on *(key, values)* pairs that are serialized within the geometry data. The JavaScript code has specific scene processors to parse the generated 3D files, retrieve and interpret these values. These data are specific to Sketchfab 's interpretation of the channels, so they were not pushed to the OSG trunk.

FBX Data optimizations The newly added maps and vertex data (texture coordinates) highlighted data duplication problems. For geometries having materials with several channels, the shared UV arrays were duplicated for each UV channel. The FBX file structure handles references and shared data, so this was due to a legacy behavior of the plugin. The duplication was fixed by keeping track of the internal references within the FBX and restoring them while building the OSG output.

The FBX plugin also appeared to duplicate materials. The way the plugin read the FBX file was not efficient. A FBX file is structured as follow :

[Illustration : SCHEMA FBX nodes, main hierarchy of a FBX file]

To understand where the duplication was made, I had to take a look on how FBX deals with data and nodes. Within FBX, a geometry may have several materials (contrary to OSG which doesn't support this feature). A FBX Node which contains a geometry also contains a list of references of scene materials. At this step, the plugin generated a material for each reference found in the FBX node. If the scene contains only one material that is used by several FBX nodes, the plugin generated an OSG material for each node, that is the origin of the duplication issue.

To avoid unnecessary memory consumption and make the material edition more efficient, I recast the way the OSG plugins handles FBX material.

Materials are now generated at the beginning of the parsing according to the scene data. In fact, the used materials are referenced within the FBX scene description before being referenced by the nodes. The changes made on the FBX plugin at this step were important enough to worth a refactoring. This part of the plugin is now cleaner and more modular.

Normal mapping and Tangent spaces The goal of the normal mapping is to simulate normal perturbations of a 3D surface with new (fake) normal values that are encoded within a texture. In fact, normal maps are commonly stored as RGB image where the RGB component correspond to the X, Y and Z coordinates respectively, of the surface normal.

When rendering using normal mapping, the normal texture is mapped onto the face according to the UV coordinates of the geometry. Image's data is not used to define the color at a given pixel, but the color values are converted into a normal vector which is used instead of the existing normal to compute lighting.

Tangent spaces are computed for each vertices and are used to apply normal mapping effects on geometries. Their data is used to convert normal map's data from face's space to model space, to be then converted to world space and used in lighting calculations.

[Illustration : triangle, normal map, drawing and result or][Illustration : normal map, RGB normal, tangent space and normal map result with lighting (2x2)] To build the tangent space of a given vertex (that is the interpolated through the face), we need a normal, a tangent and a binormal vector. Even if normals are handled by the majority of the 3D file formats, this is rarely the case for tangent and binormals, except for a few ones, such as FBX.

In the Sketchfab's pipeline, tangent spaces are systematically recomputed for each geometries in order to support the existing normal mapping (or to allow the further application of the normal mapping), but it is useless when we can get the data from the 3D file.

(Tangent, Binormal, Normal)

After retrieving these tangent and binormal vectors, tangent spaces can be *compressed* by merging both in a single 4D vector. The binormal vector is usually recomputed by cross product at the shading step (for performances) using the normal and the tangent vectors. The purpose of using a 4D vector is to store the tangent data in the three first component and use the fourth to indicate the sign of the binormal, in order to be able to choose between the two possible results of the cross product.

I added the tangent and binormal retrieval to the FBX plugin, and the conversion into 4D vector. To allow more flexibility on their retrieval and conversion, I added two option so that the user can disable the tangent and binormal parsing, and also disable the use of a single 4D vector (tangent and binormals are so stored a two different vertex attributes arrays)

OBJ

Obj is one of the most commonly used 3D file format, since it is human readable and very easy to write and parse. Developed by Wavefront Technologies, it contains the basic geometry data (positions, normals, texture coordinates and vertex colors), materials and geometry groups, but all scene or transform data are ignored. For each object, a first part gives the vertex data, and the second part describes the geometry by referencing them.

[Illustration : obj screenshot + plane]

Materials are also given within .mtl files. A single .mtl file can define multiple materials with :

Ka : ambient color (RGB) Kd : diffuse color (RGB) KS : specular color (RGB) + Ns : specular exponent (float)

The MTL also supports multiple illumination models, configurable by setting values using the field "d" or "Tr" depending on the implementation. It also support referencing textures maps (map_Kd, . . .) and handles textures parameters.

OBJ are straightforward to read and write, so some vendors don't hesitate to alter it or add custom contents. For example, zBrush, a well known sculpting tool, exports OBJ files with a custom layer of data for its polypaint and masking data.

[Illustration : zBrush icon] ZBrush is a famous digital sculpting tool developed by Pixologic, widely used in the Video game and movie industry. It provides a very rich set of tools to manage 3D objects as if they were real sculpting materials, which is more adapted for biological structures such as characters or animals. 3D sculpt doesn't have to deal with scene graph or matrices since the model is most of the time composed of a single big mesh, for which OBJ is adapted.

zBrush proposes a polypaint tool which allow to paint directly on a 3D mesh. Instead of painting onto a texture and mapping it into the model, the color data is directly stored for each vertex. During a sculpting session, the mesh is constantly modified, and it topology is often rebuilt by *remeshing*. The workflow is not adapted for the use of a texture because during a *remeshing* operation (that homogenize and simplify the structure of the mesh without altering its look), UV are lost.

So instead of using a texture, vertex colors are more commonly used in zBrush. Masks are also used to improve the painting by modifying the way the paint is applied to the surface. These data added when exporting an OBJ file from zBrush, in order to be saved for further use.

[illustration : screenshot zBrush vertex colors]

Even if the data is very specific, there are a lot of zBrush artists that upload obj models on Sketchfab, so handling the colors is worthing.

Primitive support for PLY

PLY (for Polygon File Format) is very similar to OBJ. Their structure is very adapted for 3D scanning and both are mostly used in this area.

Sample of ply file : [Illustration : screenshot of ply file]

One of the principal lack that suffered the OSG PLY plugin in OSG was that it only supported triangles primitives. When a model had quads or polygon primitives, the output model was rendered as a point cloud(consisting on printing the vertexes, without building faces between). Some 3D scans and sculpts were uploaded using PLY file and were rendered as point cloud.

As for obj, I improved the ply plugin to manage these unsupported types of primitives to allow the result to be more accurate with the source. I also worked on it to make it more flexible about certain properties since they can be some delta from one file to another.

Blender development

Blender is a professional free and open source software which provide a wide set of tools for creating animated films, visual effects, 3D models for printing or for interactive 3D applications and video games. It as a huge community with very active forums, tutorials and independent developers who contribute to its improvement and provide addons to be used with.

Blender is also scriptable through a Python API. I spent the second part of my internship working with Blender and writing scripts to both fix and improve the support of some 3D files.

Blender's specific pipeline

To process the .blend files in Sketchfab, we simply use blender calling it in command line. Command line calls allow to run Blender in background mode and run scripts. This allows to automatize operations on 3D data, created from scratch with a script or coming from a 3D file, and output the result in the same way.

The first run script is used to convert (if needed) specific blender materials into classic ones, to be handled by OSG. After that, the exporter script is run and output a .osgt file The first step deals with the shader nodes (materials represented by graph). Its role is to flatten these materials to make it compatible with the current exporter version and also the internal structure of OSG files. Material flattened, a second instance of Blender is run with the exporter script, which convert all the relevant data and outputs an osg file (*.osgt*) which is then processed.

Fixing Blender processing fails

The stats showed that a lot of .blend files were not processed correctly and most of them were causing the processing to fail. I was assigned to the different tasks that had a link with blender, in order to find a solution. The Sketchfab's Blender exporter works with a majority of files but doesn't handle some of them which have a specific structure. In some case, due to the specifications of the scene graph saved into the file, the processing failed. The fail occured because no file was output by the exporter. The difficulty in this part was that the processing logs were not very accurate, so I needed to dig up using Blender and looking for the configuration of the scene graphs in order to find commonly unsupported patterns.

The interface is very different of what I was used to use in 3DsMax, but the concepts are pretty the same. A scene graph is composed of objects (name, transform) linked to object data. The object data is in fact the active part of the object, which defines it's type (such as mesh, light or skeleton) and have the corresponding set of parameters. Blender uses a scene graph representation where the data can be shared between nodes.

Encoding errors

The first issues were related to object names that were misencoded.

Since the version 2.5, Blender used Python 3.x instead of python 2.5 which was the case before. The API was migrated to be run with Python 3 including all the new data structures. For retro-compatibility purpose, Blender also has to deal with *.blend* files that come from older versions.

The encoding problem that the current sample files had was due to the fact that the newest Blender API had to deal with files that were created using the oldest version. Contrary to Python 2 which handled a lot of encodings, Python 3 encodes all the strings as sequences of Unicode characters, so the way names' characters are managed differs between the old and the new versions of blender.

The sample I had was made using a old version (2.46) and contained objects whose name had latin-1 characters. It is file when reading and managing the data with this version, but in the case of Sketchfab, all blender files are managed using a newer Blender version (2.70). Due to the new string representation by Python 3 (which is run in parallel with the new version of the C++ Blender core), scripts are not able to deal with the misencoded names.

Since in Blender, assets are managed through dictionaries whose keys are actually the element's names, it is impossible to access objects that have non Unicode characters.

The only solution that worked for this was not really efficient, but allowed to solve the issue. The script tries to access the name property of all objects, catch the Unicode error if any and rename the object with a default, unicode name. I didn't find any other way to do it since every access has to be performed through the Python API.

Then, the fix was extended to all the Blender assets. This first issue concerned some isolated cases. I worked on more common issues related to some scene graph configurations that were not handled by the Blender exporter.

Visibility check

By opening a bunch of failing blender files, I saw that some users tried to upload blender files that have no visible objects.

When reading and parsing the *blend* scene, the exporter actually cares about the visibility of the objects. It makes sense since the user doesn't want the hidden objects to be rendered in Sketchfab. If an object is not visible in the current scene, it is just skipped. If the entire scene has no visible objects, no data is loaded and no *.osgt* file is output.

Blender has two ways to modify the visibility of an object in a given scene. The first is to set the object on the active scene to set it visible, since only the active scene is shown in the viewer (in the case of a file with several scenes).

The first way is to manage the visibility of the object within the scene graph, using the "eye button" which shows or hide the object in both the viewer and the renderer. The second way is to manage visibility using the scene layers. Layers are used to regroup objects, which allows to hide or show entire parts of the scene to facilitate scene edition. Layer can be simultaneously activated and their object are all visible.

[Illustratio : layers UI, eye icon Blender]

The user actually recieves an error message telling him that its file can't be processed correctly. This is not very accurate and can refer to a Sketchfab internal error. A better solution is to be able to tell the user that its *.blender* file was not processed because it doesn't have any visible object.

Instead of having to make the processing fail in this case, we preferred to detect empty scenes, set all its objects to visible and return a warning message to the user to explain what was detected and what was changed in its scene. A second visibility check if done after that and ends with an error if the scene is physically empty.

Hidden armatures

The only output I had for blender process fails was the error of the empty output. I had to open files within blender and look at their data and the structure of the model they contained to try to find a common pattern. A lot of them were very nice models, and the scenes were quite complexe and used a lot of elements. I found that each of them used an Armature object so I began to check that were actually these objects and if they were responsible of the crash.

To apply non destructives modification on objects (mostly on meshes), Blender uses modifier stacks. Both the viewer and the renderer take into account the modifications of the stack, but the data is internally conserved unchanged. Armature are a type of modifier which allow to deform a mesh according to a skeleton object. Armature are used for rigging, a technique allowing set the mesh into a given pose.

[Illustration : model without pose and with pose]

Armature object are often hidden from the scene, as the user uses it to determine a pose and not to render it (bones are only rendered to be set, they have no rendering purpose). This was a problem when going through the exporter.

When a mesh has an *Armature* modifier applied to, it references the previously mentioned object, which contains the skeleton data. The problem here was that *Armature* objects were not visible so ignored by the exporter, but when reading a geometry that referenced it through it's modifiers, it threw an exception and Blender was closed.

Armature's data were retrieved and saved in the output file since OSG handles animations and rigging. For Sketchfab, animations are not useful so we can skip them and avoid to write all this data in the output file.

To fix it, I firstly discarded the Armatures in the visibility check. It fixed the crash but was not sufficient, output geometries were not affected by the armature modifier.

Flattening modifiers on geometries

The blender exporter has a parameter to decide whether the modifiers had to be applied (flattened) or not. By default, this parameter was set to true in the Blender command line call, but armatures were discarded since they had to be written in the output file. Sketchfab doesn't need this data, but the goal was to render the model as it appeared in Blender, so modifiers simply needed to be applied in the geometry.

Avoiding to write skeleton data was a specific behavior for Sketchfab, so It has to be done separatly from the main code. When calling Blender with command line, scripts can be piped to be executed sequentially, so I wrote the new code in a separated script. This *pre-process* script had the responsibility of preparing the data to be exported for Sketchfab, and resolving all the problematic patterns mentioned before. [Illustratino Scema preprocess]

To simplify the blend data before writing it, we decided to flatten all the modifiers stacks on the concerned geometries. With this, the model is fixed into its final form and appears as expected on Sketchfab. The modifier application was a bit tricky and we needed to handle several cases of disabled modifiers, and other conditions that made Blender to silently cancel the modifier application.

At the end, the *pre-process* script gave good results, fixing almost 75% of the failing models. The script was a solution to fix without spending much time on it, keeping Sketchfab specific code apart. The next step is to update and refactor the main exporter to make it handle all the previously mentioned case, but I was asked to work on a project more primary.

Kerbal Space Program

[Illustration : Kerbal Space Program] One of the last projects I worked on during this internship was the support of files coming from Kerbal Space Program, a spatial simulation game. Kerbal Space program is a space flight simulator developed by squad using Unity3D, released on June 2011. KSP is still in public beta development but has already a huge community and reached the top 3 best sold games on Steam (currently only through the early access program). The game was made in a such way players can create and import their own 3D modeled parts into the game to build their spaceship. A lot of mods and packs are shared between the users to improve the game or propose new parts to extend the game experience.

Sketchfab is a very good alternative for these users to show their work and share it in the KSP's forums. We had a lot of requests in the exporter section from KSP users who wanted to share their work and working on this may make a lot of new productive users to join the community and use the platform.

A KSP member developed a Blender plugin to manage the import and the export of the parts into Blender, that Sketchfab integrated in its pipeline.

[Illustration : KSP -> blender -> osgt -> SKFB][Illustration : parts]

The current version of the exporter is limited to the support of parts only, but doesn't not allow to manage entire crafts. It was included within the Sketchfab pipeline to allow users to upload their custom or handmade parts and share them with the KSP community.

As said, KSP is a very good opportunity for Sketchfab to get new users, so the Sketchfab community team contacted the developer of the importer to propose a collaboration. With my previous work on Blender, I was assigned to this task and began to work on a full spaceship importer. The structure is pretty simple : parts are given within .mu files which are actually imported in Sketchfab, and ships are set by a .craft which contains data to puts parts together. With the help of some explanations from the developper and some samples found on the web, I started to work on. A this moment, I had the experience and the understanding of Blender I acquired during the previous works. The most difficult part was the lack of data I had about craft structure and parameters. I only focused on the data that are relevant for static 3d rendering, but some details were not very clear and searching on the web didn't helped much more, so I had to dig it blindly.

To render the ship, I only needed the data relative to the transform (position, rotation, scale) since the parts are already read by the existing code. I had to differentiate data that was internal to the game and data that concerned the 3D models.

[Illustration : part composition -> craft assemblage]

Structure of the script

Input The question of the input was important since we didn't have any informations about the copyrights of the KSP data we need to render a spaceship. Before choosing an input structure, we needed to know if parts were sharable, or if we needed to only take into account custom parts (that would not be very efficient since spaceships are not fully built with custom parts) [Illustration : zip with sketchfab.craft + repertories with craft]

Output Since parts are output as blend file, I was more interesting and time saving to continue working with blender and its Python API to do the job.

Process After having taken a look around KSP data and what was needed, I established the main process of importing a craft file:

I. Get all the given parts : check the directories to list all the available parts

II. Parse the craft and get the useful data - Get the ship name - Craft parts that have no given sources (.mu) must be skipped

III. Get the prefabs (usef parts), read the .mu files and generates the corresponding blender objects (prefabs)

IV. Loop on the craft's parts list :

    1. duplicate the prefab (without duplicating the data, linked=true)
    2. set the right transform
    3. Set the name (unique) of the object
    4. Put the generated object as children of the main object (with the name of the ship)
    5. Delete all the prefabs

V. Write and process the .blend file.

Blender part I spent some time to understand the behavior of Blender's operators, to know which operator applies on selected objects, which one applies only on active object, and the software context management.

After having defined the main structure of the script, I had to spend time on Blender's documentation. For each operation, I checked which solutions were possible, looking for the most optimized and quick way to do it. Blender was very useful to manage data thanks to its API and its internal function, but it has also some drawbacks : the API is very close to the user interface and it has some constraints. Scripts need to manage active objects and object selection in order to apply operators on it. At the beginning, I had to deal with the concepts of active objects and with object selection and deselection.Operations are based on selected or active objects which can have a lot of side effects (like unwanted object duplication), and operators need a specific Blender context to be applied (and throw an exception if it is not the case).

KSP part In KSP, the data is packed within nodes, which made it very easy to read. The .craft uses this structure so the parsing part was almost easy to write. The first expected input was a .zip archive containing a .craft file and a set of directories containing the data of each part. Each part is composed of a .cfg file where the part meta data (parameters) is given, with a value referencing the mesh file. A mesh is a .mu file which is given with one or more textures with a *.mbm* extension. From a part to another, the file names are the same, which can cause the data to be incrementally overridden in blender. The meshes were not problematic, since the name of the imported object was correctly set. The problem occurred with the materials and the textures, so texture and materials names needed to be set accordingly to the last imported object they belong, before importing the next one.

[Illustration : structure of a .craft file][Move or delete this :] #### Data duplication For Sketchfab, the data duplication is a very recurrent problematic, because it leads to bigger and more complex scene graphs which are very slow to traverse and manage in the JS side and drop down the FPS at the rendering (and this is very frustrating). Moreover, web browsers have 3D data restrictions, and client's hardware is not always "performant" enough to deal with all the data. In the case of editable data like the materials, it can make the edition very hard and fastidious to do and impact the user experience. The pipeline uses a lot of data and scene graph optimization like geometry merging and compression, which need to traverse several times the scene graph. These operations are useful but can be very slow and increase significantly the processing time in the case of a very complex and large scene graph.

Working on processing tests

In order to avoid regressions or breaking existing behavior by side effects while merging a new Pull request, Sketchfab uses processing tests. For all the work I made, in whether OSG or exporters, I also created tests to be run with all the existing ones in order to validate that my code doesn't break anything and works as expected (by asserting the expecting results). They will also be useful for further changes to check that these behavior are not affected. For example, It was very useful when I had to make huge refactors of the OSG plugins.

In this context, each pull request I submitted to Sketchfab's repositories had a corresponding one on the tests repository.

Tests structure

Basically, a test is composed of a model which is processed normally, followed by a set of assertions that check that the files were generated correctly. Assertion are used to check both the file existence, and the processing logs that are output. The whole processing outputs logs and warnings into a 'processed.log' file, which contains all the detail of what was done during the entire process. The relevant part of this data is extracted and written into a JSON file, which is read by the front end code.

Testing behaviors and logs For testing purpose, this file is read and assertions are applied on it to validate or invalidate the processing tests. The messages that are logged into are used to check processing behavior. For example, when running the smoothing visitor on a geometry during the cleaner step, a warning is written into processed.log and reported into the JSON to be parsed. The presence of this warning message testifies that we passed into the cleaner and applied the normals generation because they were not consistency. Stats and textures related data are also written for the same purpose. The other role of this file is to have the data to be able to report more accurate warnings to the user when the front end will handle this feature.

Testing 3D data A lot of tests are also made on the 3D data, running assertions on the output *OSGJS* file. OSGJS is a 3D file format coming from the OSG.JS WebGL framework, whose purpose is to offer an *OpenSceneGraph-like* toolbox to interact with WebGL using JavaScript. To make assertion on the 3D data, this file is read using a Python script (JSON parser) which allows to retrieve the relevant data using parameters:

```
- Filter : the path that leads to the wanted JSON node

- having/where : allow to retrieve nodes that fulfill a given query. *Where* only keeps node

- select : allows to refine the selection and only return the interesting part.
```

Here is an example of the test I wrote for the *blender-preprocess.py* script :

```
# Test the blender preprocess script
# The sample contains an armature, parent of three geometries:
#  - A cube having a name with latin-1 characters and two modifiers, invisible in the curren
#  - A cylinder visible but in the layer 4 (so invisible in the current scene layer)
#  - A torus and a Cone sharing the same material
# It also contains two cubes having two booleans, whose operand are the cone and the cylinde
# The whole scene has only one material defined.
# We check that the cube is renamed, the invisibility of the scene is detected and all model
# We also check that meshes are duplicated (to allow modifier application) and that the Arma
# We finally count the number of (named) materials to ensure that we don't duplicate it

process_test_model "${model_dir}/preprocess-test.blend"
assert_grep "SUCCESS" "${url_dir}/processed.log"
assert_meta_contains "warning.generic" "Meshes were duplicated to allow modifiers to be appl
assert_meta_contains "warning.generic" "1 entities having misencoded names were renamed"
assert_meta_contains "warning.generic" "The scene has no visible objects"
assert_meta_contains "warning.generic" "All the hidden meshes were set to visible"
assert_meta_contains "warning.generic" "Error while applying disabled modifier Armature (ren
local geometry_count="$( ./json_parser.py --filter osg\\.Geometry --having '{"Mode":"TRIANGI
assert_equal "${geometry_count}" "6"
local material_count="$( ./json_parser.py --filter osg\\.Material --select Name --count ${ur
assert_equal "${material_count}" "2" # The root material + the shared scene material
```

External activities

During the internship, I also had some activities around Sketchfab's community. For example, I had the opportunity to participate in some meet-ups with artists and other 3D start-up that were interesting in both cultural and human aspects.

Moreover, always outside of my engineering job, I managed the print orders Sketchfab had through the 3D Hubs platform. 3D Hubs allows to find owners of 3D printers in a given geographic zone and order 3D prints. Sketchfab was registered as one of these owners (hubs) and received several orders during the 6 months I was there. I had to download from the website and then prepare, run the print and check with the client to deliver the product. 3D printing became something very interesting for me, and I really appreciate what it brought to me. In parallel, I participate in community events where I made 3D scans of people. I went to the 3D print show which took place in Paris in October, to represent Sketchfab, scan people and also get informations about that was actually done in the 3D printing area. Finally, during my spare time, I created a few models for Sketchfab, to be used for Christmas or New Year cards.

Summary of the results

OSG

The work I made on the plugins made the rendering issues decrease significantly on Sketchfab. The huge part of STL models used to be printed are now rendered without any issues. Models with normal data consistency issues are now handled and automatically corrected within the pipeline to be rendered correctly (this concerns about [part of smoothed models] models).

FBX files are know better supported by Sketchfab and are much more easy to manage. Wireframes are kept, and the majority of the used maps are retrieved and automatically set within the editor, which improves significantly the user experience. Moreover, data are more efficiently retrieved since we don't duplicate it anymore. Scene graphs are lighter, since a lot of data is now shared. Materials are easier to edit, and tangent spaces are not systematically recomputed.

At a lower level, OSG plugins were refactored and updated with a cleaner code. The updated versions were submitted to be merged into the main repository so users can enjoy the new supported features.

Blender

The work I made for Blender has very interesting results too. We reprocessed all the failing models we had using the new version of the Blender pipeline, and almost 75% are know correctly processed. Blender has a huge community and blender artists often make very nice models, so this better support is interesting for Sketchfab.

Kerbal Space Program

The KSP importer is usable and is able to import efficiently a complete spaceship without any trouble. We are waiting for news from the author of the mu importer and from the KSP community to have a bunch of models to test before communicating on it. At its final step, the importer will bring a lot of new users on Sketchfab and make the community to growth. I may have the opportunity to continue working on it to make it more efficient and maybe fix edge cases that might be highlighted with the coming samples.

Observations and personal gain

Work and workflow

The first thing that I learnt during this internship was the gain in efficiency the use of the different tools provides. I didn't used such tools during my last internship, so this one was very interesting on this points. Using them is very helpful to adopt good techniques and good behavior to have when working on a project with several developers. Code review are a very good source of learning, and more generally, the workflow gives a good overview of the main steps that required for a good project managing.

Communication

Working at Sketchfab also highlighted the importance of the communication, and how to communicate efficiently while working. Even if it looks straightforward, asking the good questions and being understandable can be difficult. Moreover, communicate is very helpful for efficiency : it allows to gain time by checking that the provided solutions are *approved*, and also to have advices from other developers, who can for example highlight some specific cases that we don't necessarily have in mind.

Importance of tests

Another thing that I learnt by working at Sketchfab was the importance of tests in the software engineering process. In a few case, they helped me to spot side effects that I didn't though about while modifying some snippet of code in OSG. I also learnt how to write good and efficient tests.

Coding

As mentioned before in this report, code reviews were very helpful for me. I received a lot of advices on *how to do things clearly and efficiently*, with a lot of good coding behaviors to have (and not to have). I also really appreciate to have discovered and began to learn Python language through my work with Blender. I also learnt when and how to refactor efficiently an existing code, and the importance of refactoring in general.

3D

Finally, I gain proficiency in 3D engineering trough the work I made at Sketchfab, as I expected at the beginning. I also learnt a lot from the conversation I had from other team members. Moreover, the environment was very interesting and helpful for discovering and learning. Knowing and understanding real-time rendering mechanism brought me a lot of skills and techniques for my personal 3D modeling practicing. I am able to anticipate and know what I have to do to reach a given result. Finally, I had the opportunity to hear about advanced concepts, such as Physical Based Rendering, which are currently appearing and implemented within a lot of software, or more advanced real time rendering features. These will be very helpful for my further jobs.

Conclusion :

Looking back to the beginning of the internship, I really feel that I have made a huge step further in 3D software engineering. Even if I regret a little bit to not had worked on advanced features, because of the lack of skills in the area, I acquired a strong basis in 3D software development, that I am now able to deepen by myself or during my future jobs. Sketchfab has taught me the good way to think, to communicate and to approach efficiently a problem. I also acquired good coding and working rules, and have an overview of what a such project can involve. I also really appreciate participating in the Sketchfab community and being involved in the environment and the company's culture. This experience was really enriching in both professional and personal aspects. To conclude, I think I reached my goals as intern.