Le paramétrique au service de l'architecte dans le processus de conception de plans

Mémoire de master

Concevoir et construire l'architecture

Arthur ROULAND

2021

Activités et Instrumentation de la conception

Encadré par

François Guéna
Joaquim Silvestre
Anne Tüscher

ensa paris la villette

Remerciements

Je tiens à remercier mes directeurs de mémoire : Monsieur François GUENA, Madame Anne TÜSCHER ainsi que Monsieur Joaquim SILVESTRE. Je les remercie pour leur encadrement, leurs conseils et leurs encouragements qui m'ont permis de réaliser ce mémoire de recherche.

J'aimerais remercier particulièrement François GUENA pour m'avoir initié à l'architecture et la modélisation paramétrique.

Je voudrais remercier mon ami Ewen COSSEC pour son aide précieuse sur l'écriture du script informatique ainsi que mes parents, Stéphane et Stéphanie ROULAND, pour leur aide à la relecture de ce mémoire.

Enfin, je remercie toutes les personnes qui ont pu participer à mes recherches et à l'élaboration de ce mémoire.

Sommaire

Sommaire détaillé

0

Avant propos

Arthur ROULAND

Étant étudiant en Master d'architecture dans le séminaire Activités et Instrumentation de la Conception à l'École Nationale Supérieure d'Architecture de Paris La Villette (ENSAPLV), et ayant été très intéressé et inspiré par les recherches du chercheur Stanislas Chaillou, j'ai voulu travailler sur la génération de plans. A l'origine, l'idée de ce mémoire était d'axer mes recherches sur une optimisation de plan. L'objectif aurait été d'obtenir un agencement d'espace permettant d'optimiser les plans et de réduire ainsi au maximum les circulations entre les logements mais également au sein même des logements. Cependant, en réfléchissant sur ce sujet et en faisant l'état de l'art, j'ai orienté ma recherche dans le domaine de la génération de plan à l'aide de l'Intelligence Artificielle, dans la continuité des recherches de différents étudiants de l'ENSAPLV. Après quelques semaines de recherches et d'apprentissage dans ce domaine, je me suis confronté à plusieurs problèmes majeurs. En effet, les recherches en Intelligence Artificielle nécessitent des connaissances en informatique et notamment en code informatique que je n'ai pas et que j'ai eu beaucoup de mal à assimiler. J'ai alors décidé d'aborder ce sujet d'une façon différente. Les recherches en génération de plans avec l'Intelligence Artificielle avancent chaque année, cependant chaque chercheur se confronte à un problème de taille qui est l'accès aux banques de données de plans. Les intelligences artificielles nécessitent ce que l'on appelle une phase d'entraînement durant laquelle les programmes s'exercent tout seuls à reconnaître des catégories d'objets avant de pouvoir les générer en s'appuyant sur ces bases de données. J'ai alors réorienté mes réflexions dans ce sens afin de m'inscrire dans la continuité des recherches effectuées dans le domaine. Ainsi, l'objectif de ce mémoire sera d'explorer une manière possible de générer des plans afin de pouvoir constituer une banque de données conséquente qui pourrait être utilisée par la suite en génération de plan avec l'Intelligence Artificielle.

Schéma de fonctionnement d'un programme de Deep Learning (Intelligence Artificielle)

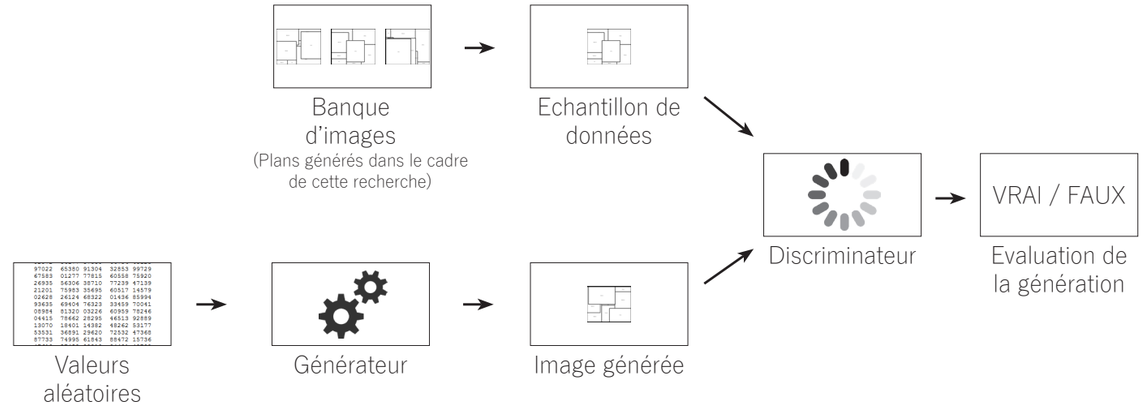


Figure 1
Schéma de fonctionnement d'un programme de Deep Learning
© Arthur ROULAND

1

Introduction

Arthur ROULAND

Les outils de conception en architecture ne cessent de se développer. Ainsi, grâce à l'outil informatique de plus en plus performant et diversifié, de nouvelles possibilités s'offrent aux architectes en matière de forme, de calcul et de génération de formes. L'objet de ce mémoire porte sur l'utilisation des outils paramétriques dans la génération de plans.

Mon travail consiste donc en une recherche visant à développer un programme qui pourra générer des plans très diversifiés afin de les utiliser par la suite dans le domaine de la recherche en machine learning ou encore en deep learning. Ce programme réalisé sur Grasshopper permettra de générer une multitude d'agencements qui répondront à des critères de dimensions et de surfaces des différentes pièces à obtenir. Les plans se verront attribuer un score qui permettra leur évaluation afin que le programme puisse en juger la qualité.

Dans un premier temps nous ferons un point historique de l'évolution de la conception. Nous verrons brièvement les étapes par lesquelles les architectes sont passés afin d'arriver à la situation que nous connaissons aujourd'hui.

Nous aborderons ensuite le contexte dans lequel ce mémoire s'inscrit. Nous verrons dans cette partie la problématique que nous pouvons extraire en fonction des questionnements concernant ce sujet. Cela nous permettra alors d'établir une méthode expérimentale pour proposer une réponse à cette dite problématique.

A l'issue de cela, nous ferons l'état de l'art. Il sera question ici de s'intéresser à ce qui se fait actuellement dans le vaste domaine de la Conception Assistée par Ordinateur (CAO ou CAD : Computer Aided Design) et plus précisément en architecture. Nous regarderons les différentes méthodes utilisées ainsi que les démarches des chercheurs ayant travaillé dans ce domaine afin d'avoir connaissance des possibilités. Cela nous permettra également de voir les limites de chacune des méthodes et proposer ainsi une nouvelle approche pour répondre à notre questionnement.

Ensuite nous verrons les supports informatiques utilisés afin de réaliser cette expérience présentant ainsi leurs caractéristiques respectives ainsi que leurs domaines d'application. Puis nous ferons un bilan de l'évolution de la recherche, les étapes par lesquelles je suis passé avant d'arriver au résultat final ainsi que les problèmes que j'ai pu résoudre. Cette expérience n'est pas l'unique façon de générer des plans mais une des multiples manières d'y arriver. Nous verrons donc les choix que j'ai pu faire afin de résoudre certains problèmes qui ont, par la suite, eu un impact sur le résultat des plans générés.

Une fois l'évolution de la recherche expliquée, nous regarderons le résultat de l'expérience de ce mémoire qui nous permettra de répondre à notre questionnement. Le fonctionnement du programme final sera développé en détail dans cette partie afin d'en faciliter la compréhension. Enfin, nous pourrons regarder un échantillon des plans générés.

Pour terminer, nous verrons la notice d'utilisation du programme. Afin de s'inscrire dans une démarche de recherche, je souhaitais rendre accessible le fonctionnement du programme afin qu'une tierce personne puisse se l'approprier mais également que quiconque puisse faire fonctionner le programme simplement afin de générer une banque de plans.

2

Point historique

Arthur ROULAND

Durant la Renaissance, un nouveau métier s'affirme : l'architecte. Bien que beaucoup de personnes ont exercé cette pratique bien avant la Renaissance, c'est à cette période, que le métier se définit à part entière dans le rôle qu'il occupe encore aujourd'hui. L'architecte était alors chargé de concevoir le projet mais aussi d'en assurer le bon déroulement tout au long de sa réalisation. Depuis bien longtemps, les architectes se sont munis d'outils pour les aider à concevoir. Tout d'abord de simples stylos, équerres, règles et compas. Cependant, une innovation va venir chambouler l'avenir de la manière de concevoir l'architecture et l'architecture elle-même : l'invention de l'informatique. L'informatique est apparue progressivement et a évolué au fil du temps. L'invention n'est pas un processus linéaire, cela résulte de multitudes de tests d'améliorations parmi lesquelles il résulte ce qui a été sélectionné comme amélioration viable ou intéressante. On retrouve également beaucoup de tentatives ratées ou bien d'améliorations qui n'ont pas su trouver leur utilité à leur époque. Ce serait vers 1834 que Charles Babbage aurait conçu la première machine analytique : un calculateur mécanique programmable basé sur un système de cartes perforées reliées à un organe de commande. Herman Hollerith, invente quant à lui une machine électromécanique permettant de stocker des données sur une carte perforée en 1880. Au fur et à mesure de l'évolution de l'informatique, l'ordinateur a été inventé jusqu'à arriver à ce que nous connaissons aujourd'hui. Tout d'abord, dans les années 50, l'ordinateur a été développé dans un but militaire tout comme de nombreuses inventions. Il a peu à peu été adapté jusqu'à arriver à un usage civil, celui que nous connaissons aujourd'hui avec nos ordinateurs, smartphones, tablettes bien d'autres. Les premiers outils de Conception Assistée par Ordinateur (ou CAO ou CAD en anglais) apparaissent et ce n'est réellement que dans les années 70 qu'ils se développent afin que les architectes puissent les utiliser. Cependant, le CAD peut être appréhendé de multiples façons. Tout d'abord utilisé afin de faire sur ordinateur ce que les architectes faisaient avant sur papier en dessin en deux dimensions : un ensemble de traits ou de courbes qui créent des formes jusqu'à arriver à du CAD en trois dimensions. Jusqu'alors, ce CAD servait à dessiner, à créer, à concevoir sur un outil informatique, mais l'ordinateur n'intervenait pas réellement dans le processus de conception. Aujourd'hui, cette utilisation du CAD est globalement maîtrisée par l'ensemble du corps de métier. Cependant, la Conception Assistée par Ordinateur ne se limite pas à un simple changement de support de dessin et de conception.

Avec l'apparition du paramétrique dans les années 80, le CAD s'ouvre à un nouveau champ des possibles. Dans cette branche du CAD, l'ordinateur joue un rôle majeur dans la notion de conception. En effet, dans le domaine du paramétrique, les objets virtuels ne sont pas dessinés par les architectes mais leurs paramètres sont décrits étape par étape. Ainsi, une boîte n'est pas simplement qu'un volume dessiné arbitrairement dans un espace virtuel mais devient un quadrilatère avec un grand côté de longueur x et un petit côté de longueur y, extrudé suivant la normale de ce plan d'une longueur z. Cette boîte a un point d'origine dans l'espace qui est défini par l'utilisateur et celle-ci est orientée dans les trois plans selon un angle également défini. Ainsi, chaque étape du processus de conception est définie, précisée et paramétrée par le concepteur qui ne va que très peu dessiner voire pas du tout. L'ordinateur va ainsi calculer l'ensemble des contraintes et paramètres définis lors du processus de conception afin d'arriver un modèle 2D ou 3D respectant l'ensemble de ces contraintes. Cette façon d'utiliser le CAD permet la création de nombreuses formes, objets complexes très difficiles voire impossibles à dessiner sans avoir recours à l'utilisation de l'informatique. Certains architectes et designers en ont même fait leur signature. L'utilisation de logiciels paramétriques dans l'utilisation du CAD est relativement récente et donc la plupart des architectes n'utilisent pas cette façon de concevoir. Cependant, la conception paramétrique n'est pas l'unique forme où la machine intervient amplement dans le processus de conception. Depuis quelques années, une nouvelle branche de la conception architecturale se développe grandement : l'utilisation de l'Intelligence Artificielle.

3

Problématique

Arthur ROULAND

Dans le domaine de la recherche en génération de plan, on peut retrouver plusieurs logiques expérimentales permettant d'arriver à ce résultat. Parmi elles, on trouve notamment la génération de plan en utilisant un programme de machine learning ou de deep learning qui sont des branches du domaine de l'intelligence artificielle. Cependant, comme nous avons pu l'évoquer précédemment, ce type de programme est fait pour apprendre et nécessite donc une phase d'entraînement. En somme, on va donner à un programme d'intelligence artificielle des informations sur lesquelles il va s'entraîner. Le but de l'entraînement peut être par exemple de reconnaître une certaine typologie de plan. Une fois cette phase d'apprentissage terminée, lorsque le programme ne se trompe que très peu (le taux d'erreur est convenable comparé au taux de réussite), alors il est possible d'utiliser cet outil afin de générer des formes aléatoires qu'il va reconnaître. Ainsi le programme ne va nous proposer que des résultats qu'il reconnaît et donc générer des plans correspondant aux critères que l'utilisateur lui soumet.

Cependant, afin de générer des plans répondant à certaines exigences, la phase d'entraînement nécessite une grande quantité d'informations de départ sur laquelle le programme va s'exercer. Certains organismes répertorient des plans, ayant un aspect homogène et les mettent à disposition comme Rakuten par exemple. Il semble d'ailleurs que ce soit cette banque de données que le chercheur Stanislas Chaillou a utilisée pour le développement de son intelligence artificielle. Cependant, ces plans fournis par Rakuten répondent à des standards asiatiques, la phase d'entraînement est donc influencée par ce paramètre et les plans alors générés le sont pour leurs similarités avec ces données initiales.

L'objectif de cette recherche est alors de créer un programme permettant de générer des plans simples, qui répondront à des critères paramétrables. Ainsi, on pourra obtenir une grande diversité de plans qui serviront par la suite comme banque de données dans la recherche en intelligence artificielle. Les plans générés seront évalués en fonction de certaines de leurs caractéristiques qui leur sont propres pour pouvoir les noter et les classer dans des catégories.

Pour l'élaboration de cette expérience, j'ai souhaité utiliser des outils les plus simples possibles afin de permettre à une plus grande diversité de personnes d'avoir accès à cette recherche afin de la modifier, l'améliorer et ainsi pouvoir générer des plans adaptés à leurs attentes. L'objectif est de générer des formes simples mais avec un très large spectre de possibilités pour permettre l'élaboration d'une banque de données la plus diversifiée possible

4

Etat de l'art

Arthur ROULAND

4.1 - Introduction de l'état de l'art

Cette partie qu'est celle de l'état de l'art va nous permettre de prendre connaissance de l'avancée du savoir dans le domaine de la génération de plans. Cependant, comme nous avons pu le voir précédemment, nous pouvons appréhender ce sujet de plusieurs façons. Nous allons donc tout d'abord faire l'état de l'art de la génération de plan à l'aide de l'Intelligence artificielle, puis nous ferons l'état de l'art de ce sujet dans les domaines des automates cellulaires, des systèmes multi-agents ainsi que celui de l'allocation spatiale.

4.2 - L'intelligence artificielle

L'intelligence artificielle est l'ensemble de théories et de techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine. Ces techniques sont aujourd'hui largement utilisées dans divers domaines utilisant la technologie comme nos smartphones, tablettes et ordinateurs. Certains chercheurs et architectes se sont alors penchés sur l'adaptation et l'utilisation de l'intelligence artificielle dans le domaine de l'architecture et donc de la génération de plans.

4.2.1 - Stanislas CHAILLOU

Dans le domaine de la recherche en génération de plans pour l'architecture, les travaux de Stanislas CHAILLOU sont très souvent mentionnés. Ce chercheur a présenté en Février 2020 une conférence se tenant au Pavillon de l'arsenal à Paris intitulée : « AI & Architecture ». Cette conférence d'une vingtaine de minute disponible sur internet présente les travaux que Chaillou à pu faire ces dernières années. Évoluant essentiellement dans le domaine de l'intelligence artificielle, cet architecte chercheur a alors développé un programme de Deep Learning permettant de générer des plans d'architecture tout à fait convaincants par leur aspect, leur ordonnance etc. Dans la vidéo du Pavillon de l'Arsenal, il explique l'élaboration de son programme et les étapes par lesquelles il est passé afin de développer un tel dispositif.

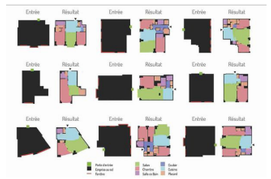


Figure 4.2.1.a
Travail de Stanislas CHAILLOU
Source : cahiers-techniques-batiment.fr

Figure 4.2.1.b
Photo de Stanislas CHAILLOU
Source : pavillon-arsenal.com

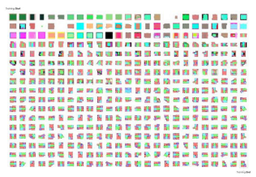


Figure 4.2.1.c
Travail de Stanislas CHAILLOU
Source : paulvanderlaken.com

4.2.2 - Jean Raphaël PIQUARD

Le mémoire de Jean-Raphaël PIQUARD a également participé à l'élaboration de mon mémoire. Cet élève de l'École Nationale Supérieure d'Architecture de Paris La Villette (ENSAPLV), s'est intéressé à la génération de plan à l'aide de l'intelligence artificielle. Dans son mémoire on peut voir qu'il a tout d'abord commencé par générer des chiffres entre 0 et 9. Il a ensuite réussi à générer des formes géométriques simples comme des carrés, des triangles, des losanges etc. Après ces étapes de recherches, il a adapté son programme à la génération de formes noires dans un espace blanc. Quelque chose de très intéressant émerge de cette recherche car les formes générées par le programme de Jean-Raphaël Piquard peuvent appartenir à certaines catégories identifiables et reconnaissables mais son programme génère également ce qu'il appelle des chimères. Ces dernières n'appartiennent à aucune des catégories ou pourraient appartenir à plusieurs catégories en même temps. Ce résultat est intéressant car grâce à ce processus de recherche, il est possible de générer des plans chimériques appartenant à des typologies qui sortent de l'ordinaire.

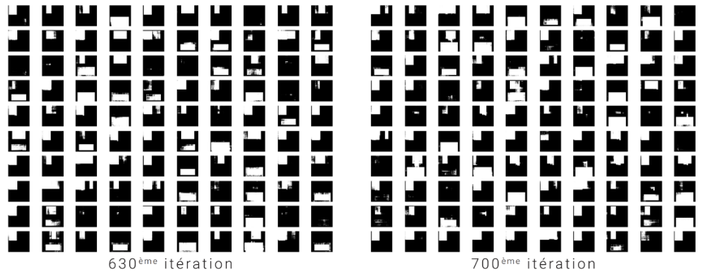


Figure 4.2.2.a
Exemple de génération
Source : Mémoire de Jean-Raphaël PIQUARD

4.2.3 - Alex SALINI

Alex SALINI a lui aussi été étudiant à l'ENSAPLV. Durant son Post-Master Recherche en Architecture, il a rédigé en Septembre 2020, un carnet de recherche sur "L'utilisation d'un réseau adversarial antagoniste (GAN) dans la création de plans d'architecture". Tout comme Jean-Raphaël PIQUARD, il s'est intéressé à l'utilisation de l'intelligence artificielle et s'est inspiré des travaux de Stanislas CHAILLOU. Cependant, il semble qu'Alex SALINI se soit directement attaqué à la génération de plans sans élaborer d'étapes dans son programme. La seconde différence importante est la banque de données. Il semble avoir fait le choix d'une banque de plans beaucoup plus précise, d'une qualité de résolution beaucoup plus élevée quitte à avoir une banque beaucoup plus réduite avec 250 plans (Jean-Raphaël PIQUARD utilisait des images de 50 par 50 pixels). Le résultat de son programme d'intelligence artificielle n'était pas concluant par rapport au résultat attendu mais les générations qu'il a pu faire demeurent très intéressantes et permettent tout de même de produire une multitude de propositions.

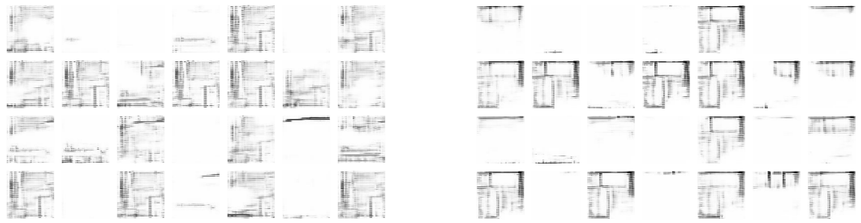


Figure 4.2.3.a
Exemple de génération
Source : «L'utilisation d'un réseau adversarial antagoniste (GAN) dans la création de plans d'architecture»
Alex SALINI, 2020

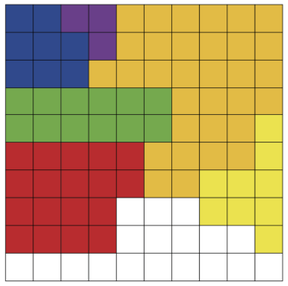
Arthur ROULAND

4.3 - Les automates cellulaires

Un automate cellulaire consiste en une grille régulière de « cellules » contenant chacune un « état » choisi parmi un ensemble fini et qui peut évoluer au cours du temps. Cette méthode permet de générer en deux dimensions ou en trois dimensions un assemblage de cellules. Certains chercheurs se sont penchés sur ce sujet afin de l'adapter à la génération en architecture, qu'elle soit à l'échelle de plans, de bâtiments ou de quartiers de villes. L'utilisation des automates cellulaires est très intéressante car elle permet de paramétrer chaque unité à l'aide de l'informatique. Les formes ainsi générées répondent donc à des critères définis et variables dont on peut connaître les paramètres.

4.3.1 - Nathan BEYLER

Nathan BEYLER a également été étudiant à l'ENSAPLV et a rédigé son mémoire de recherche de master "Utiliser et transformer des critères en esquisses de plan" en 2019. Cet étudiant s'est donc intéressé à la génération de plan en se basant sur un principe d'automates cellulaires dans l'optique de soumettre des propositions aux architectes. Nathan BEYLER ne s'inscrit pas dans la continuité de la recherche en intelligence artificielle mais souhaite plutôt développer des esquisses de plans en prenant en compte différents paramètres tels que la lumière, le son et la chaleur. Sa recherche lui a permis d'élaborer une manière de diviser un plan en le transformant en une grille de cellules et en divisant cette grille avec un automate cellulaire afin de définir différents espaces de couleur.



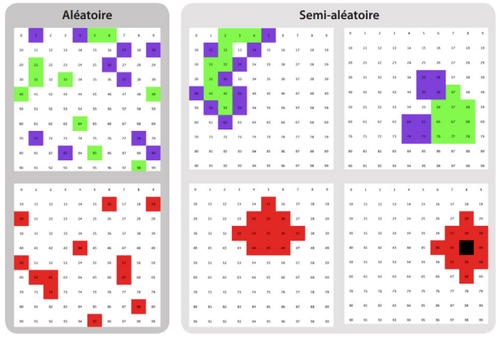
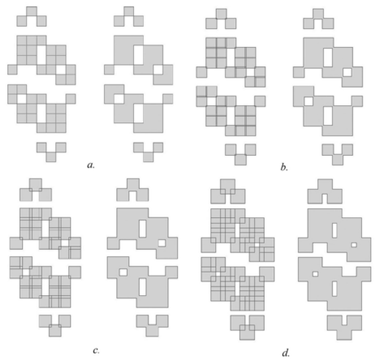


Figure 4.3.1.a
Exemple de génération
Source : «Utiliser et transformer des critères en esquisses de plan»
Nathan BEYLER, 2019

4.3.2 - Robert J. KRAWCZYK

KRAWCZYK Robert J. est un professeur et chercheur du College of Architecture de l'Illinois Institute of Technology à Chicago aux Etats-Unis. Son papier de recherche intitulé "Architectural Interpretation of Cellular Automata" publié en 2002 porte comme son nom l'indique sur l'interprétation architecturale des automates cellulaires. Ce travail consiste à générer des relations entre des cellules sur un plan horizontal mais aussi vertical dans l'optique de produire des formes en trois dimensions ayant des relations. Les formes ainsi générées sont par la suite interprétées de différentes manières en utilisant différentes formes qui font varier le résultat final. On voit ici que les automates cellulaires ne sont qu'un support à la créativité. Les relations dans les deux plans sont générées selon des paramètres mais le résultat laisse place à différentes interprétations esthétiques et spatiales.



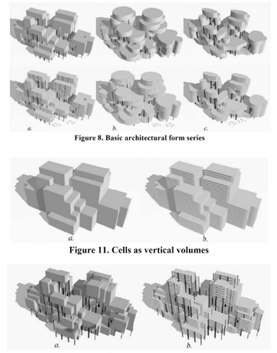


Figure 8. Basic architectural form series

Figure 11. Cells as vertical volumes

Figure 12. Cells with retained growth

Figure 4.3.2.a
Travaux de Robert J. KRAWCZYK
Source : «Architectural Interpretation of Cellular Automata»
Robert J. KRAWCZYK, 2002

4.3.3 - Christiane M. HERR & Ryan C. FORD

Christiane M. HERR est une enseignante et chercheuse à Southern University of Science and Technology à Shenzhen en Chine. Ryan C. FORD est quant à lui, un architecte Néo-Zélandais. Ces deux personnes ont collaboré sur un papier de recherche intitulé "Adapting Cellular Automata as Architectural Design Tools" publié en 2015. Tout comme KRAWCZYK, ces deux chercheurs tentent d'appliquer les automates cellulaires à la conception architecturale en modifiant les règles des automates cellulaires afin qu'ils correspondent à celles de l'architecture. Les formes générées dans cette recherche ne restent que formelles et l'interprétation est ensuite faite par l'humain. Le programme élaboré ne semble pas permettre de transformer ces formes en plans architecturaux. Cependant, le but de ce papier de recherche est plutôt de démontrer qu'il est possible d'appliquer ce système à l'architecture, de démocratiser ce support d'aide à la conception et de le considérer comme un réel outil dans le processus de conception.

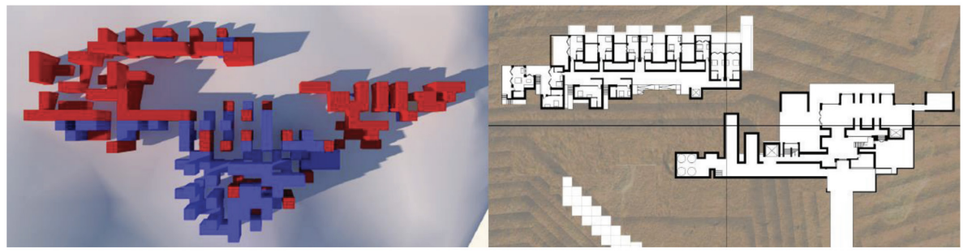


Figure 4.3.3.a
Travaux de Christiane M. HERR & Ryan C. Ford
Source : «Adapting Cellular Automata as Architectural Design Tools»
Christiane M. HERR & Ryan C. Ford ,2015

Arthur ROULAND

4.4 - Les systèmes multi-agents et l'allocation spatiale

Les systèmes multi-agents sont des systèmes composés d'un ensemble d'agents (un processus, un robot, un être humain, une fourmi etc.), actifs dans un certain environnement et interagissant selon certaines règles. Ce procédé se rapproche de celui des automates cellulaires mais ne se base pas forcément sur l'utilisation d'une grille. Les architectes et les chercheurs ont utilisé cette méthode afin de pouvoir générer des plans ordonnés répondant à des critères. Ces derniers sont souvent liés aux relations spatiales mais aussi fonctionnelles qu'entretiennent les espaces entre eux. Ce système appliqué en architecture se rapproche du principe d'allocation spatiale. Celle-ci est la disposition informatique des pièces dans un plan. C'est le processus de détermination de la position et de la taille de différentes pièces dans un espace à deux dimensions, en fonction des exigences de l'utilisateur et des contraintes topologiques et géométriques. L'allocation spatiale va généralement travailler dans un espace restreint, une forme visant à être agencée selon des paramètres contrairement au système multi-agents dont la forme va être générée par la création du plan.

4.4.1 - Zifeng Guo

Zifeng Guo est un chercheur de l'Ecole Polytechnique Fédérale de Zurich en Suisse. Dans ses travaux intitulés "Evolutionary approach for spatial architecture layout design enhanced by an agent-based topology finding system" publiés en 2017, il aborde la génération en architecture en se basant sur un système multi-agents. Un système multi-agents est défini par wikipédia comme étant "[...] un système composé d'un ensemble d'agents (un processus, un robot, un être humain, une fourmi etc.), actifs dans un certain environnement et interagissant selon certaines règles". A l'aide de cette méthode, ce chercheur dispose des espaces qui vont interagir entre eux de plusieurs manières : l'attraction, la répulsion, l'échange et la compression. Grâce à cette méthode, il arrive à imbriquer et organiser les espaces entre eux en plan mais également dans un univers virtuel en trois dimensions. Par la suite, il simplifie sa méthode en se basant sur un système de grille permettant d'obtenir des résultats très convaincants.

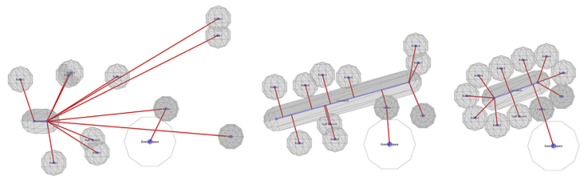


Figure 4.4.1.a
Travaux de Zifeng Guo
Source : Evolutionnary approach for spatial architecture layout design enhanced by an agent-based typology finding system
2017

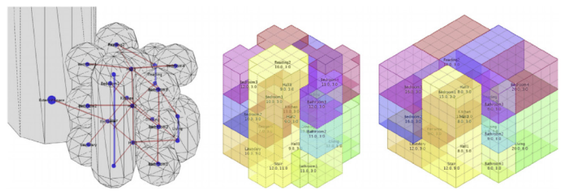


Figure 4.4.1.b
Travaux de Zifeng Guo
Source : Evolutionnary approach for spatial architecture layout design enhanced by an agent-based typology finding system
2017

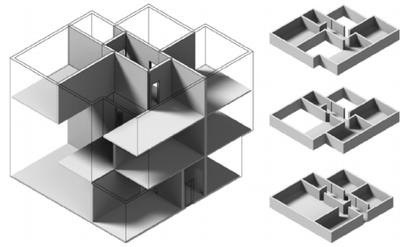


Figure 4.4.1.C
Travaux de Zifeng Guo
Source : Evolutionnary approach for spatial architecture layout design enhanced by an agent-based typology finding system
2017

Le second article de ce chercheur auquel je me suis intéressé s'intitule "Generated Building Layout". Cette recherche est également basée sur l'utilisation de systèmes multi-agents mais uniquement dans un environnement à deux dimensions. Dans ces travaux, il spécifie les noms des pièces ainsi que les relations qu'elles entretiennent avec les espaces avoisinants. Ces espaces sont alors disposés dans un rectangle dans lequel le plan final doit s'inscrire. L'ordinateur propose alors un agencement en disposant des cloisons et des ouvertures en façade afin de constituer un plan.

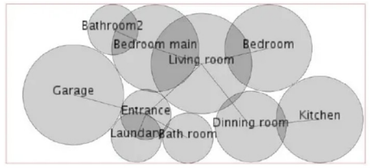


Diagram of one-floor layout.

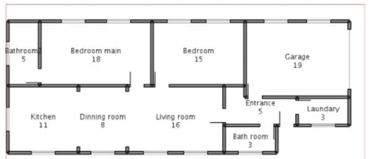


Figure 4.4.1.d
Travaux de Zifeng Guo
Source : Generated Building Layout

4.4.2 - Silvio CARTA

Silvio CARTA est un enseignant chercheur ayant travaillé à l'université de Cagliari en Italie, à l'université de Rotterdam et à l'université de Delft aux Pays-Bas. Il travaille désormais à l'université de Hertfordshire au Royaume Uni. Ses travaux portent sur l'organisation autonome des plans à l'aide d'outils informatiques. Deux d'entre eux m'ont particulièrement intéressé : "Self-organizing Floor Plans" réalisé à l'université d'Hertfordshire et publié le 27 Mai 2020 et "Self-Organising Floor Plans in Care Homes" réalisé en Mai 2020 avec Stephanie St Loe également chercheuse à l'université d'Hertfordshire. Le premier travail de ce chercheur tend à une optimisation du plan à chaque génération. Ainsi, le meilleure agencement d'espaces est utilisé en entrée pour la génération suivante dans l'optique d'améliorer le résultat à chaque itération. Les premiers plans générés sont très intéressants car ils ne répondent pas aux standards d'architecture. Les formes et leur disposition sortent de l'ordinaire et cette approche permet d'explorer de nouvelles partitions. La seconde partie de cette recherche ressemble à un système multi-agents où l'utilisateur va venir disposer des espaces en spécifiant les connexions de chaque pièce. Le programme va venir générer une proposition de plan en respectant ces conditions.

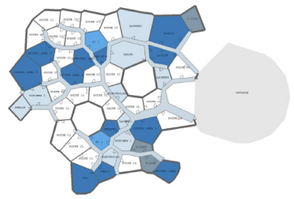


Figure 4.4.2.a
Travaux de Silvio CARTA
Source :Self-organizing Floor Plan
27 Mai 2020

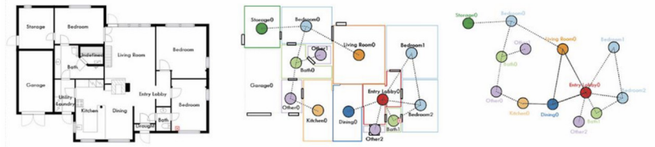


Figure 4.4.2.b
Travaux de Silvio CARTA
Source :Self-organizing Floor Plan
27 Mai 2020

Arthur ROULAND

Le second article auquel je me suis intéressé est "Self-Organising Floor Plans in Care Homes" réalisé avec Stephanie ST LOE. Ces deux travaux de recherche ont été publiés en Mai 2020 et portent sur le même sujet. Cependant, ce second travail intègre également le champ de vision depuis certains points du plan. Ainsi, les connexions qu'entretiennent les espaces peuvent traverser les pièces et évoluer selon le cheminement d'un potentiel usager.

Figure 10. Viewing cones of agents in the simulation. Agents gathering to the kitchen.

Figure 4.4.2.c
Travaux de Silvio CARTA
Source :Self-organizing Floor Plan in Care Homes
Mai 2020

4.4.3 - Buiding Generator with Geometry Nodes

Blender est un logiciel de 3D gratuit et communautaire. Ainsi, les utilisateurs peuvent créer des plug-ins, des add-ons que tout le monde peut télécharger et utiliser. Durant l'année 2021 est sortie une nouveauté sur le logiciel Blender : le Building Generator with Geometry Nodes. Cet add-on gratuit permet la génération de bâtiments entiers. En paramétrant des textures, des formes de façades, de planchers, de structures et de percements, le logiciel est capable de générer en quelques secondes des bâtiments. Pour l'instant, il semble que l'add-on soit utilisé dans le but de créer les formes extérieures des bâtiments au détriment de la partition du plan. Cependant, cette méthode pourrait très facilement être appliquée à la génération de plans.

Figure 4.4.3.a
Exemple de génération
Source : Instagram @antoinebagattini
15 Novembre 2021

4.4.4 - Krishnendra SHEKHAWAT

Krishnendra SHEKHAWAT est un chercheur diplômé de l'Indian Institute of technology de Delhi (2008) et de l'université de Genève (2013). Il s'est notamment intéressé à la question de l'allocation spatiale en architecture qu'il définit comme ceci : "Space allocation is the computational arrangement of rooms (spaces) in a floor plan. In other words, it is the process of determining the position and size of different rooms in a two-dimensional space, according to the user's requirements and topological and geometric constraints". En somme nous pourrions le définir en français comme la disposition informatique des espaces dans un plan en déterminant la position et la taille de chaque espace du plan. Il a réalisé de nombreux travaux de recherche à ce sujet comme un article nommé "Automated space allocation using mathematical techniques" publié en Avril 2015. Dans cet article, ce chercheur a créé une méthode afin de disposer des espaces tendant à respecter les dimensions d'un rectangle d'or (le ratio du grand côté du rectangle sur le petit doit être égal à 1.618). Ainsi, il a divisé en 4 un plan ayant la forme d'une croix et ces 4 espaces sont alors divisés en pièces ayant également ce ratio doré.

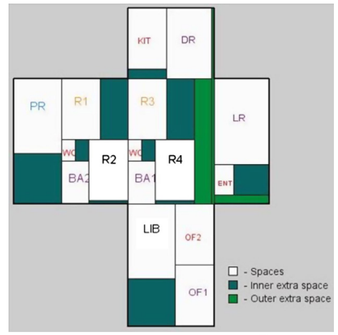


Figure 4.4.4.a
Travaux de Krishnendra SHEKHAWAT
Source : Automated space allocation using mathematical techniques
1er Avril 2015

Le second article de ce chercheur qui m'a particulièrement intéressé s'intitule "Space Allocation in Rectangular Floor Plan" et a été publié en Décembre 2012. Cette thèse réalisée pour l'Université de Genève semble être les prémisses de la recherche publiée en Avril 2015. Il y détaille sa démarche et les opérations mathématiques mises en œuvre afin de paramétrer les différents espaces ainsi que leurs proportions. Le résultat de ces recherches est très intéressant car la division et le rapport de surface qu'entretiennent les pièces entre elles lui permettent de hiérarchiser les espaces tout en créant les inscrivant dans des rectangles dorés (selon certains consensus, les proportions dorées sont les proportions parfaites que l'on peut retrouver dans le corps humain, la nature, certaines structures minérales etc…).

Figure 4.4.4.b
Travaux de Krishnendra SHEKHAWAT
Source : Space Allocation in Rectangular Floor Plan
Décembre 2012

Arthur ROULAND

4.4.5 - Finch 3D

L'agence d'architecture suédoise Wallgren Arkitekter et l'entreprise de construction suédoise BOX Bugg ont développé un plug-in de programmation visuelle sur Grasshopper. Après avoir dessiné les murs extérieurs d'un bâtiment ainsi que ses cloisons intérieures, ce plug-in permet d'agencer les espaces en dessinant du mobilier et des portes. Il est possible de faire mouvoir les cloisons et les murs, ainsi le plan s'adapte automatiquement aux nouveaux paramètres. Cet outil peut permettre d'aider les architectes à aménager leurs plans et leur proposer des solutions différentes de celles qu'ils auraient pu concevoir. Il semble cependant que ce programme ne propose qu'une seule partition de l'espace par plan et n'explore pas toutes les possibilités. De plus, les paramètres doivent être modifiés manuellement ou peuvent potentiellement être programmés par l'utilisateur. Cela reste une supposition.

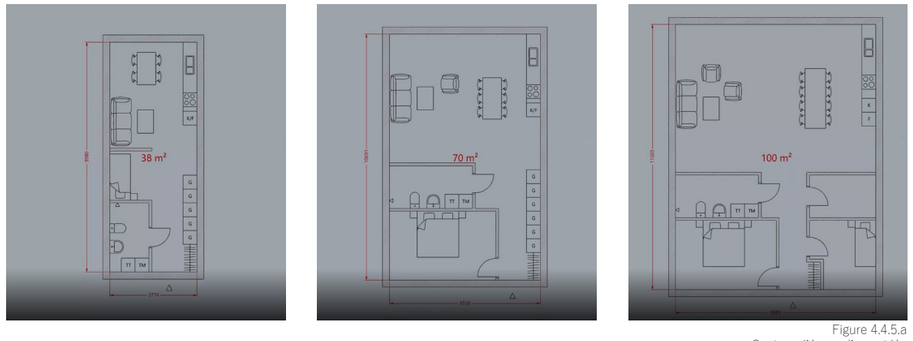


Figure 4.4.5.a
Capture d'écran d'une vidéo
Source : archdaily.com
6 Décembre 2019

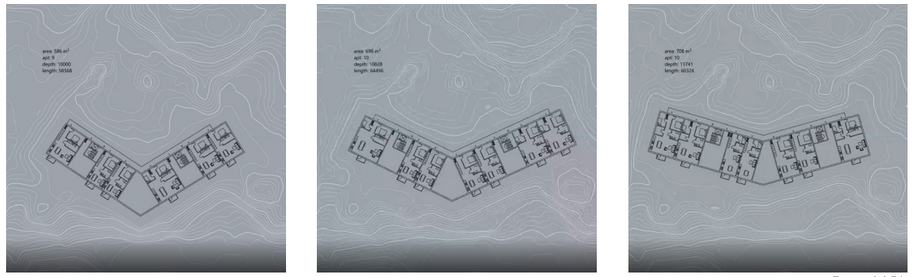


Figure 4.4.5.b
Capture d'écran d'une vidéo
Source : archdaily.com
6 Décembre 2019

4.5 - Conclusion de l'état de l'art

Pour conclure cet état de l'art, il semble que beaucoup de personnes s'intéressent au domaine de la génération en architecture. Cela peut prendre différentes formes à différentes échelles (celle d'un logement, d'un bâtiment ou d'un bout de ville). Diverses méthodes sont utilisées et il en existe certainement d'autres que nous n'avons pas pu évoquer ici. Cet état de l'art n'a pas pour but d'être exhaustif sur ce domaine mais plutôt de donner un aperçu de la situation actuelle dans le domaine de la génération en architecture. Nous avons ici pu voir différentes méthodes que sont l'intelligence artificielle, les automates cellulaires, les systèmes multi-agents ainsi que l'allocation spatiale.

Nous verrons dans la partie 5 de ce mémoire de recherche que la méthode retenue est celle des automates cellulaires. En effet, j'ai pu suivre l'année dernière les cours "TR707 : Initiation à la modélisation paramétrique" ainsi que le cours "CTID 825 : systèmes de la conception digitale" dispensés à l'ENSAPLV par François GUENA. Ces cours portaient en partie sur l'utilisation de cette méthode pour la génération de formes sur Rhinocéros 3D à l'aide du plug-in Grasshopper. Cette méthode m'est donc relativement familière mais elle est également plus accessible que certaines autres comme l'utilisation de l'intelligence artificielle qui requiert un certain nombre de connaissances en langage informatique.

5

Etapes de la recherche

Arthur ROULAND

5.0 – Les logiciels

5.0.1 - Rhinoceros 3D

Pour réaliser cette expérience, j'ai choisi d'utiliser le logiciel Rhinoceros 3D. Ce logiciel Conception Assistée par Ordinateur a été sélectionné pour plusieurs raisons. Premièrement, il permet de générer des formes très variées et complexes ce qui laisse alors une grande liberté à l'utilisateur. Il est plus libre que certains autres logiciels utilisés en architecture comme Archicad ou Revit par exemple. Rhinoceros 3D est disponible sur PC et sur Mac ce qui pourra également permettre à un plus grand nombre de personnes d'accéder aux fichiers afin de pouvoir les modifier. Enfin, Rhinoceros 3D propose un plug-in nommé GrassHopper dont nous allons nous servir tout au long de cette expérimentation

Figure 5.0.1.a
Logo de Rhinoceros 3D
Source : food4rhino.com

5.0.2 - Grasshopper

Grasshopper (GH) est un plug-in de Rhinoceros 3D. Un plug-in (encore appelé module d'extension ou module externe) est un programme complétant les fonctionnalités d'un logiciel. Le plugin GrassHopper permet de créer des modèles paramétriques sur Rhinoceros grâce à de la programmation visuelle. GrassHopper a été sélectionné car il est utilisé par certains architectes concevant des projets à l'aide de l'outil paramétrique. Cette extension permet la création de formes variées et cela nous servira donc à générer des dispositions de plans dont on connaît les paramètres, que l'on peut faire varier mais dont on peut également en extraire les données. De plus, générer des formes à l'aide du paramétrique dans Rhinoceros 3D avec GrassHopper permet de visualiser le résultat, offrant ainsi une meilleure appréhension et compréhension du programme

Figure 5.0.2.a
Logo du plug-in GrassHopper
Source : food4rhino.com

5.1 – Premières recherches avec le plug-in « Marmot »

Tout d'abord, j'ai voulu baser mes recherches sur des programmes déjà existants disponibles sur Food4Rhino afin de comprendre leur fonctionnement. J'ai donc pu analyser ce que ces programmes permettaient de générer mais j'ai également identifié leurs limites. J'ai notamment téléchargé et utilisé l'add-on « Marmot ». Cet add-on sert à générer une répartition d'espaces au sein d'un rectangle que l'utilisateur définit. En testant ce programme j'ai pu identifier des avantages et des inconvénients à baser mes recherches sur cet add-on. L'avantage aurait été la facilité d'utilisation. Marmot permet d'indiquer différentes pièces dans un plan qui est définissable par l'utilisateur. Il permet d'indiquer la surface de ces différentes pièces mais également les relations qu'elles entretiennent entre elles. C'est-à-dire que l'on peut indiquer si l'on souhaite que la pièce 1 communique avec la pièce 2. Ce programme est assez efficace, cependant, la surface que l'on accorde à chaque pièce semble être approximative sans réelle surface minimale. L'ensemble des surfaces des pièces varie afin de trouver un équilibre pour occuper l'ensemble du plan. Les différents espaces n'ont pas de longueurs minimales, et en ce sens, une chambre de 10 mètres carrés pourrait être générée comme étant un rectangle de 10 mètres par 1 mètre. Enfin, cet add-on ne semble fonctionner qu'avec une surface rectangulaire. Un test avec une surface polygonale plus quelconque a été réalisé mais le programme ne semble pas réussir à aménager le plan.

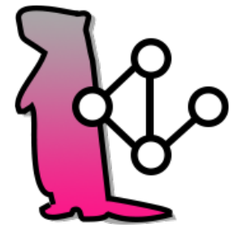


Figure 5.1.a
Logo de l'add-on Marmot
Source : food4rhino.com

5.2 – Premières recherches avec le plug-in « Magnetizing Floor Plan Generator »

En continuant mes recherches sur l'avancement de la génération de plan, j'ai trouvé l'addon "Magnetizing Floor Plan Generator". Ce programme est bien plus complet mais également plus complexe que l'add-on Marmot. Magnetizing Floor Plan Generator permet comme son nom l'indique de générer des plans. Tout comme pour Marmot, j'ai pu télécharger et essayer cet outil et j'ai identifié des avantages et des inconvénients à ce programme. En premier lieu, cette extension est plutôt complète et permet de générer des formes de plans très variées, en indiquant les différentes pièces souhaitées, d'afficher leurs surfaces respectives mais également les relations qu'elles entretiennent entre-elles comme dans l'extension Marmot. L'avantage est la diversité de paramètres sur lesquels l'utilisateur peut intervenir mais également la diversité des formes générées. Malheureusement, cet add-on est très complexe à utiliser et est très spécifique. Les données générées par ce programme sont au final peu exploitables dans le sens où il est difficile de récupérer ces données afin de les réutiliser en complétant le programme. Les surfaces des pièces sont strictement égales au paramètre prédéfini par l'utilisateur et ne peuvent pas varier. La relation qu'ont les pièces entre elles est certes très précise mais doit être changée manuellement. Dans ce mémoire, il est primordial que l'ordinateur puisse faire varier ces paramètres sans intervention humaine une fois l'algorithme lancé.

Figure 5.2.a
Logo de l'add-on Magnetizing Floor Plan Generator
Source : food4rhino.com

Arthur ROULAND

5.3 – Division de la surface avec le composant « Substrate »

Après ces tentatives d'adaptation d'un programme existant à ma recherche, j'ai donc compris que la réussite de cette dernière reposerait sur le choix de la méthode de division d'une surface en différents espaces. L'enjeu est donc de trouver une façon de diviser une surface immuable en plusieurs espaces qui eux peuvent se trouver dans un domaine variable.

Afin de tenter de résoudre ce problème, je me suis intéressé au composant "Substrate" de GrassHopper. Cet élément permet la division d'une surface en un nombre d'espaces donné. Par défaut, l'angle de cette division est aléatoire. En indiquant un angle "pi" j'ai pu obtenir une division où les espaces se rencontrent en formant des angles droits. Ainsi la surface principale a été découpée en plusieurs rectangles formant ainsi la partition du plan. En sortie de ce composant, j'ai pu récupérer ces surfaces.

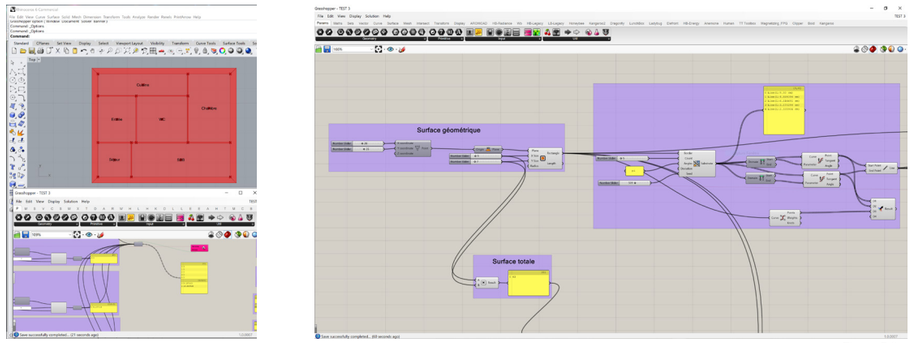


Figure 5.3.a
Illustration du composant «Substrate»
© Arthur ROULAND

J'ai alors commencé à élaborer une partie d'algorithme me permettant d'évaluer les surfaces obtenues. A ce stade de la recherche, l'évaluation de la qualité des espaces ne repose que sur la surface au sol des pièces. Grâce à ce système, il est donc possible de savoir si la superficie d'une pièce se situe dans un intervalle que j'ai pu indiquer. En sortie de cette évaluation, si la surface se trouve dans l'intervalle, alor, sa valeur sort de l'algorithme. Si sa surface est trop petite ou trop grande, alors la valeur 0 sort de l'algorithme. Les sorties de chaque pièce sont alors regroupées afin de constituer le score final.

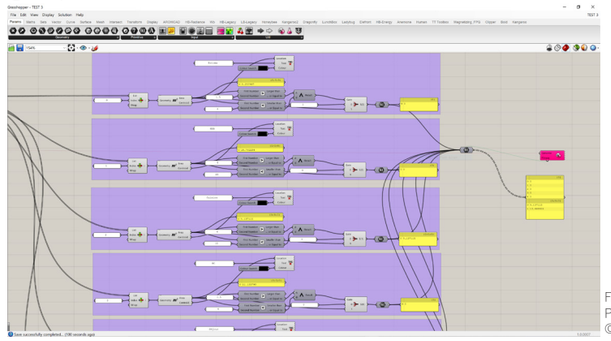


Figure 5.3.b
Premier programme d'évaluation
© Arthur ROULAND

Pour optimiser les résultats, j'ai choisi d'utiliser un module de design génératif intégré à GrassHopper : "Galapagos". Ce composant récupère une ou plusieurs valeurs et peut agir sur des paramètres que l'on définit afin de les faire varier dans l'optique d'optimiser ou de minimiser la valeur de sortie. A cette étape de la recherche, l'utilisation de Galapagos n'a pas été convaincante car le système d'évaluation donnait en sortie des surfaces et non pas des scores. Galapagos cherchait donc soit à les optimiser, soit à les minimiser.

Figure 5.3.c
Logo de l'add-on Galapagos
Source : food4rhino.com

5. 4 – Modification du système d'évaluation

Dans cette nouvelle étape, j'ai donc modifié mon système d'évaluation afin qu'il n'évalue non plus la valeur des surfaces mais si ces dernières respectent ou non les critères que je leur demandais. Pour ce faire, les surfaces étaient évaluées afin de voir si elles se trouvaient toujours dans l'intervalle de valeur prédéfini. Au lieu d'envoyer leur valeur en sortie, elles se voyaient attribuer un score : 0 ou 1. Ainsi, si les pièces respectent les surfaces demandées, elles obtiennent 1 point. Cependant, la surface ne reste qu'un indice de qualité d'espace : une chambre de 15m² peut paraître très confortable dans un logement mais si cette chambre fait 15 mètres de long sur 1 mètre de large alors l'espace n'est pas praticable. Afin de résoudre ce problème, un nouveau module d'évaluation est venu s'ajouter au premier me permettant d'évaluer la longueur des côtés. Cette partie de l'algorithme permet donc de vérifier si les côtés sont égaux ou supérieurs à une valeur donnée. Si c'est le cas, alors on va attribuer 1 point à la pièce, dans le cas contraire : 0 point.

A l'issue de cela, j'ai tenté de réutiliser Galapagos pour optimiser cette fois -ci le résultat qui sortait des modules d'évaluation. Plus le score est élevé, plus il respecte les critères que l'utilisateur demande au programme de génération.

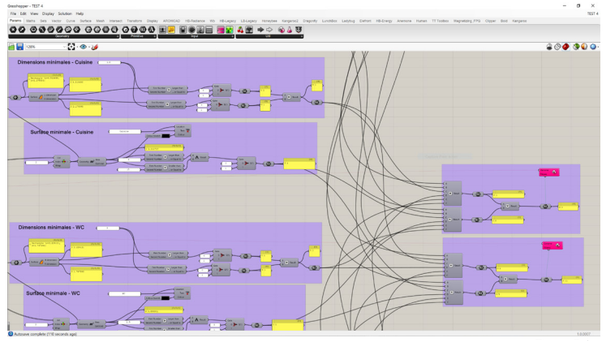


Figure 5.4.a
Système de points
© Arthur ROULAND

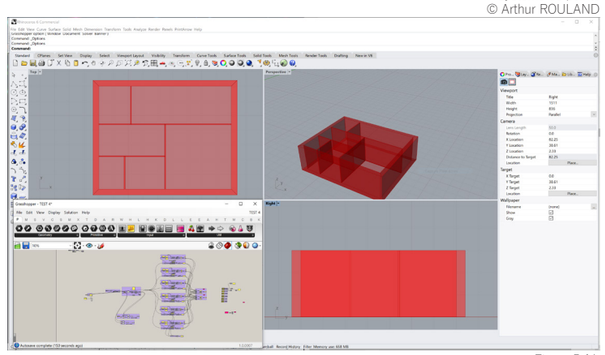


Figure 5.4.b
Premiers résultats obtenus avec Galapagos
© Arthur ROULAND

5.5 – Recherches sur la division de la surface

5.5.1 - Division en voronï

En utilisant le composant Substrate, Galapagos ne pouvait intervenir que sur un seul paramètre d'entrée : Seed (graine). Cette Seed peut être modifiée en changeant la valeur d'entrée. Cependant, aucune logique apparente ne se dessine entre la Seed 1, la Seed 2, la Seed 3 etc··· Galapagos cherche à trouver une valeur optimisée et cela n'est pas possible avec un paramètre Seed. Afin d'optimiser les chances de résultats, j'ai indiqué que ce paramètre pouvait prendre un large panel de valeurs afin d'en trouver potentiellement une qui permettrait d'avoir un résultat satisfaisant. Cependant, j'ai découvert que les Seeds n'étaient pas infinies. Les données de ces dernières se répètent. Ainsi, par exemple, la Seed numéro 17 peut être la même que la Seed 1017. Galapagos n'était donc pas en mesure d'influer correctement sur les valeurs de départ afin de chercher à optimiser le résultat de sortie.

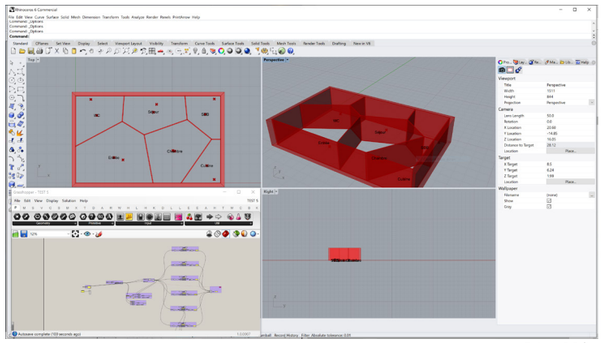


Figure 5.5.1.a
Résultat obtenus avec un système de voronoï
© Arthur ROULAND

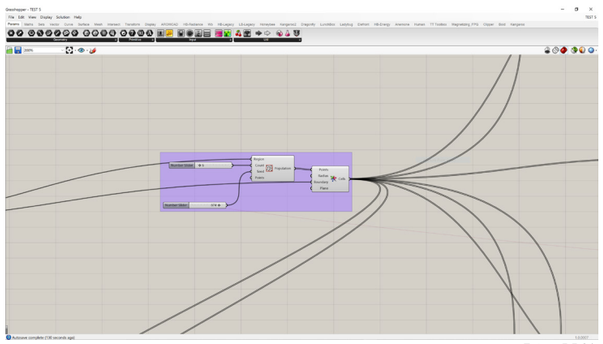


Figure 5.5.1.b
Système de voronoï
© Arthur ROULAND

Il a donc fallu chercher une nouvelle méthode de division de la surface. Je me suis alors penché sur une division en voronoï. Cette méthode permet de placer un nombre de points définis sur une surface et de diviser cette surface en cellules. Malheureusement, les différents composants permettant d'appliquer cette méthode ne permettaient pas d'obtenir des angles de 90° , de paramétrer le nombre de côtés de chaque espace formé ou encore de régler la position et la forme précises des cellules. Cette piste a donc été écartée.

5.5.2 - Division par segments

Une autre piste explorée afin de diviser une surface a été de me baser sur le Cours Transversal Intra-Domaine "Systèmes numériques de la conception digitale" que j'ai pu suivre quelques mois auparavant avec François Guéna à l'ENSAPLV. J'ai donc repris le programme que j'avais réalisé pendant ce cours et j'ai tenté de l'adapter aux enjeux de ma recherche. Ainsi j'ai pu disposer 5 points sur la périphérie du plan afin de tracer des segments venant découper la surface originale. Cela créait alors différentes zones correspondant à des pièces. Cependant, avec cette méthode, le nombre de pièces obtenues était trop grand. J'ai alors tenté d'approfondir le programme afin que certains traits puissent s'arrêter lorsqu'il rencontraient d'autres segments. Cette piste n'a pas été concluante et j'ai donc continué mes expérimentations.

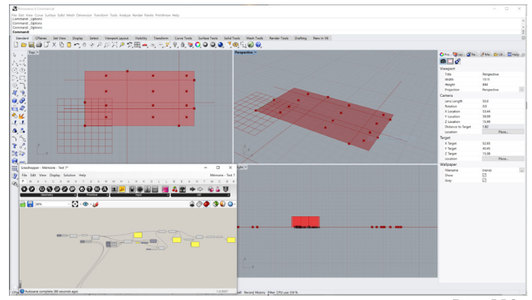


Figure 5.5.2.a
Résultat obtenu avec une division par des segments
© Arthur ROULAND

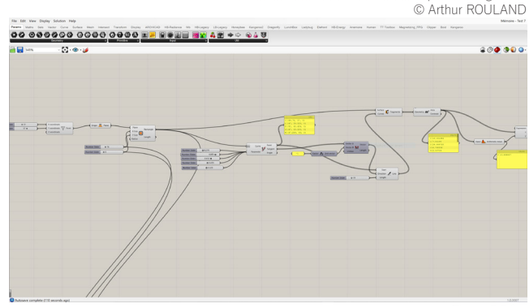


Figure 5.5.2.b
Système de division par des segments
© Arthur ROULAND

5.5.3 - Division de la surface en une grille

En continuant mes recherches sur les méthodes de division d'une surface je me suis intéressé au composant "Quad Grid" de l'add-on LunchBox de GrassHopper. LunchBox permet d'explorer les formes mathématiques, les panneaux, les structures et d'autres choses pour ensuite les représenter en 3D dans GrassHopper dans Rhinoceros 3D. J'ai tenté d'utiliser cet add-on afin de pouvoir diviser une surface rectangulaire en une multitude de carrés. L'idée était à cette étape de la recherche de grouper ces carrés/cellules afin de diviser le plan en plusieurs formes rectangulaires utilisant toutes les axes X et Y afin de former des angles droits entre les différents espaces. N'ayant pas trouvé de manières pour regrouper les cellules, cette piste a été mise de côté.

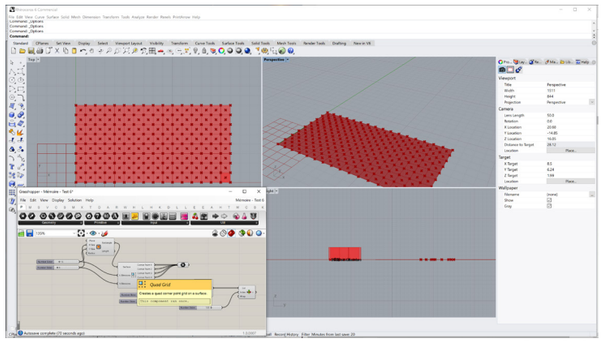


Figure 5.5.3.a
Résultat obtenu avec Quad Grid
© Arthur ROULAND

Figure 5.5.3.b
Logo de l'add-on LunchBox
Source : food4rhino.com

Arthur ROULAND

5.6 – Automate cellulaire

Durant mon Master à l'ENSAPLV, j'ai eu l'occasion de suivre deux cours dispensés par François Guéna. Ces cours 4 et 5 du CTID825 : "Systèmes numériques de la conception digitale" dispensés en semestre 8 à l'ENSAPLV portaient sur l'utilisation des automates cellulaires dans GrassHopper. Dans ce cours, il était possible grâce au plug-in "Anemone" de Grasshopper, de créer des boucles de répétition d'une opération à l'aide d'un composant codé en langage Python. Cet exercice était basé sur deux automates cellulaires "Day& Night" et "Le jeu de la vie". Dans cet exercice il était possible de changer la valeur d'une ou plusieurs des cellules appartenant à une surface. A cette valeur on pouvait attribuer une couleur afin qu'elle soit visible dans l'interface de Rhinoceros 3D. Sur ce principe j'ai décidé d'essayer de modifier ce programme afin de créer non plus 2 valeurs (une allumée et une éteinte) mais 6 valeurs (une valeur éteinte : 0 et 5 valeurs de couleur). Ainsi j'ai pu diviser une surface en une multitude de surfaces appartenant à 6 catégories. Cette méthode offre de nombreuses possibilités car la répartition des valeurs va être faite par le composant Python. De cette manière, selon le code Python que l'on créé/utilise, il est possible de régler la répartition des différentes valeurs/couleurs selon ce que l'on souhaite obtenir.

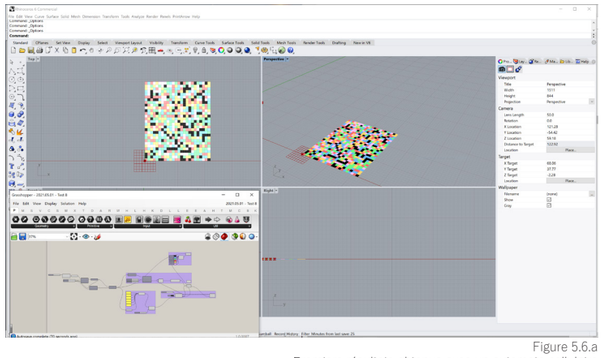


Figure 5.6.a
Premiers résultats obtenus avec un automate cellulaire
© Arthur ROULAND

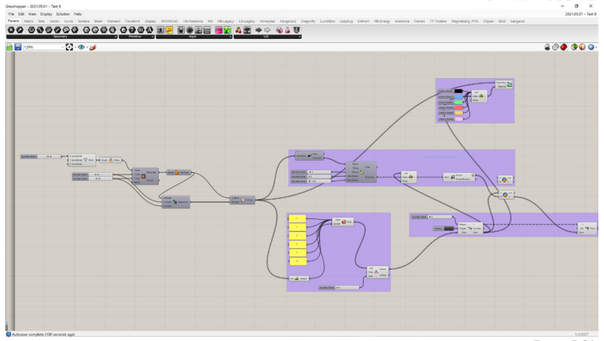


Figure 5.6.b
Programme de l'automate cellulaire
© Arthur ROULAND

Figure 5.6.c
Logo de l'add-on Anemone
Source : food4rhino.com

5.7 – Division par propagation

Dans cette méthode, la surface est divisée en une grille. A chacune des cellules de cette grille est attribuée la valeur 0 excepté 5 d'entre elles ayant respectivement la valeur 1, 2, 3, 4 et 5. Ces 5 valeurs vont par la suite être récupérées afin de former une surface correspondant à une pièce. L'utilisation de cette méthode implique un script Python ce qui a été une nouveauté pour moi. En me basant sur le cours portant sur les automates cellulaires, j'ai tenté d'écrire un programme afin que les cellules "allumées" puissent se propager à l'instar de taches d'encre. Dans ce cours, nous utilisons un voisinage paramétré dans GrassHopper qui a été ensuite envoyé dans une des entrées du composant Python. Dans cet exemple, j'ai tenté d'effectuer une propagation pour les valeurs 1 et 2 uniquement afin de voir si cela marchait et comment les valeurs réagiraient lorsqu'elles allaient se rencontrer. Comme on peut le voir sur les images ci-dessous, le résultat n'était pas très concluant.

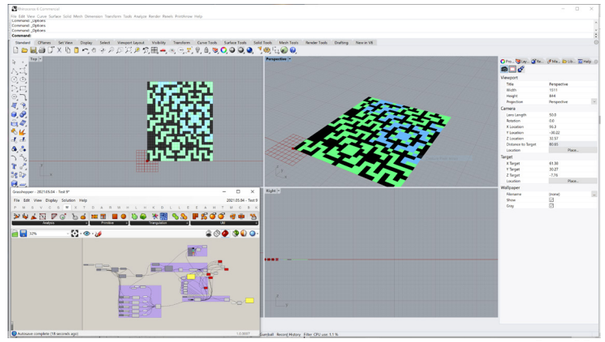


Figure 5.7.a
Premiers tests de propagation par l'automate cellulaire
© Arthur ROULAND

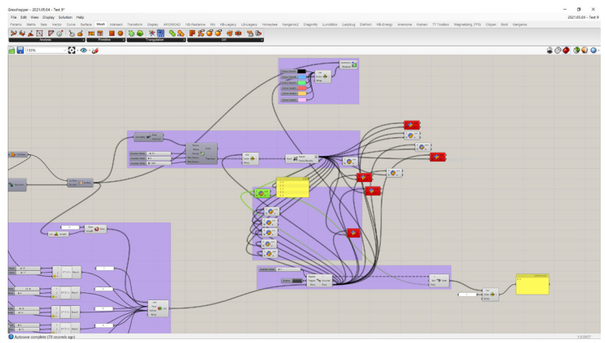


Figure 5.7.b
Programme de l'automate cellulaire
© Arthur ROULAND

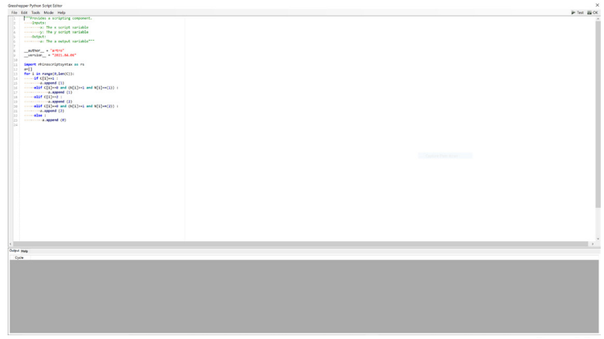


Figure 5.7.c
Premier script Python pour la propagation par automate cellulaire
© Arthur ROULAND

Arthur ROULAND

5.8 – Le script Python

La recherche repose sur la collaboration et sur l'aide. Les chercheurs se basent sur des thèses, des articles ou encore des papiers de recherche. Ils publient ensuite leurs productions afin que d'autres personnes puissent s'appuyer dessus. En m'inscrivant dans cette démarche, je me suis basé sur des écrits mais j'ai également fait appel à mon réseau afin de m'aider dans ma recherche. Mon ami Ewen COSSEC ayant effectué des études dans la programmation de jeux vidéos a accepté de m'aider à coder les commandes que je souhaitais réaliser. Grâce à son aide, j'ai pu clarifier mes intentions et approfondir ma compréhension du langage Python. Cependant, cela n'a pas été chose aisée, beaucoup d'essais ont été nécessaires avant d'arriver à un résultat satisfaisant. Nous avons, à cette étape de la recherche, réussi à propager les valeurs (et donc les couleurs) à la manière de taches d'encre comme je le souhaitais.

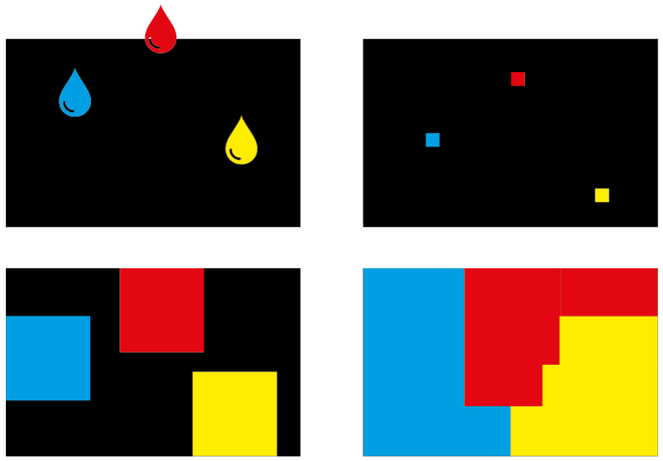


Figure 5.8.a
Schéma illustrant le système de propagation par tache d'encre
© Arthur ROULAND

Un paramètre important qui influe beaucoup sur le résultat de l'automate cellulaire est le voisinage. Le voisinage consiste à définir quels sont les voisins d'une cellule. Il existe plusieurs voisinages remarquables dont deux que j'ai pu utiliser : le voisinage de Moore et le voisinage de Von Neumann. Celui de Von Neumann va regarder 4 voisins d'une cellule. Le voisin du dessus, celui du dessous, celui de gauche et celui de droite. Le voisinage de Moore, quant à lui, va regarder ces 4 cellules mais également les 4 cellules se trouvant dans ses angles afin d'inscrire la cellule étudiée dans un carré. Afin de limiter les diagonales et tenter d'avoir, dès l'étape de l'automate, un résultat le plus rectangulaire possible, j'ai décidé de baser ma recherche sur l'utilisation du voisinage de Moore.

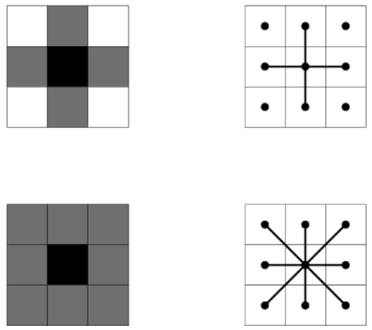


Figure 5.8.b
Schéma illustrant les voisinages de Von Neumann et de Moore
© Arthur ROULAND

Cependant, le code que nous avons élaboré à ce moment de la recherche présentait encore plusieurs problèmes. En effet, la définition de la grille est un réel sujet dans l'utilisation d'un automate cellulaire. La grille ne connaît pas ses propres limites. C'est à l'utilisateur de le lui indiquer dans le script Python. Ce problème n'avait été que partiellement réglé et les couleurs pouvaient se propager d'un côté à l'autre de la grille sans raison apparente comme on peut le voir sur les images ci-dessous. Cependant, le résultat engendré par ce code me permettait de continuer à avancer sur la suite du programme malgré ces problèmes.

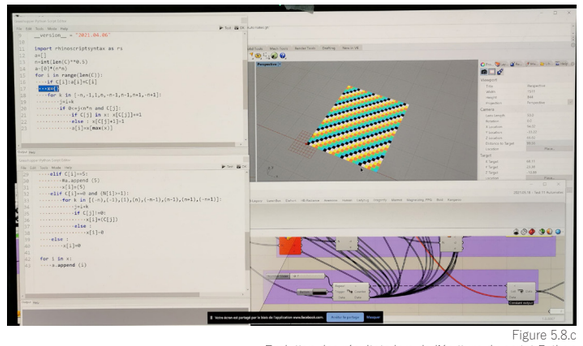


Figure 5.8.c
Evolution des résultats lors de l'écriture du script Python
© Arthur ROULAND

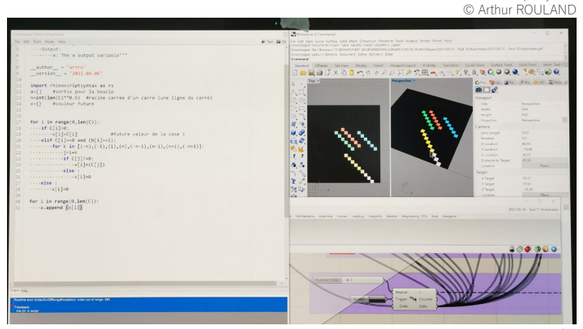


Figure 5.8.d
Evolution des résultats lors de l'écriture du script Python
© Arthur ROULAND

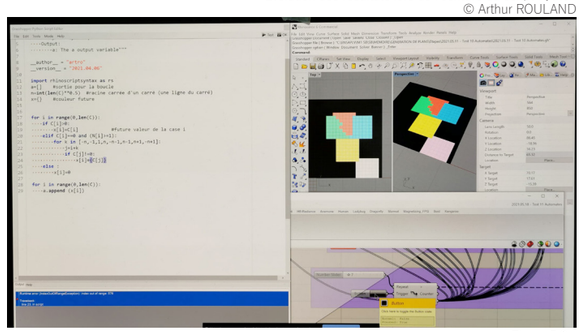


Figure 5.8.e
Evolution des résultats lors de l'écriture du script Python
© Arthur ROULAND

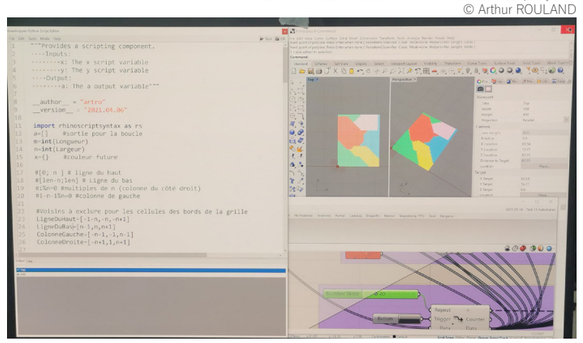


Figure 5.8.f
Evolution des résultats lors de l'écriture du script Python
© Arthur ROULAND

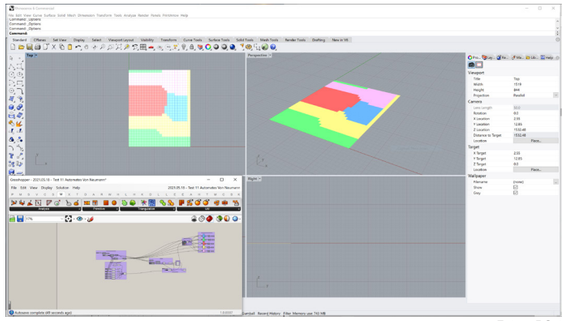


Figure 5.8.g
Résultat de l'écriture du script Python
© Arthur ROULAND

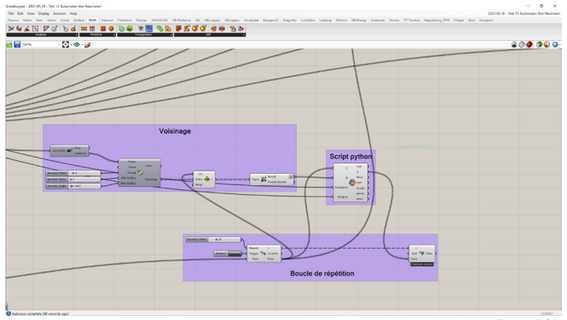


Figure 5.8.h
Script Python
© Arthur ROULAND

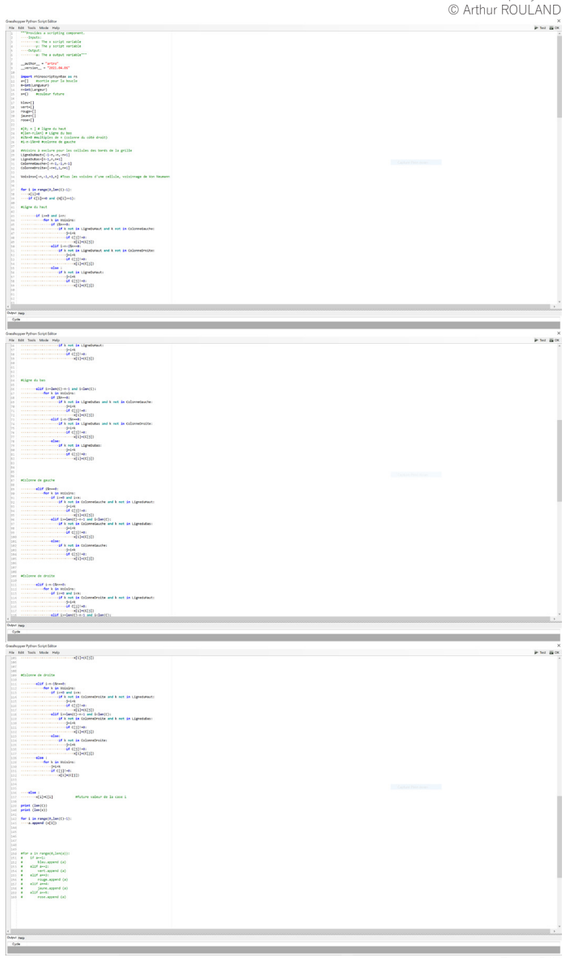


Figure 5.8.i
Script Python
© Arthur ROULAND

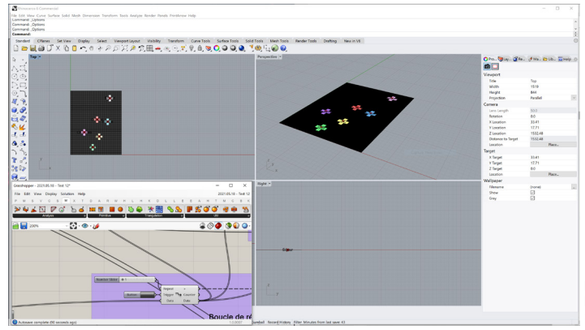


Figure 5.8.j
Itérations générées avec le script Python
© Arthur ROULAND

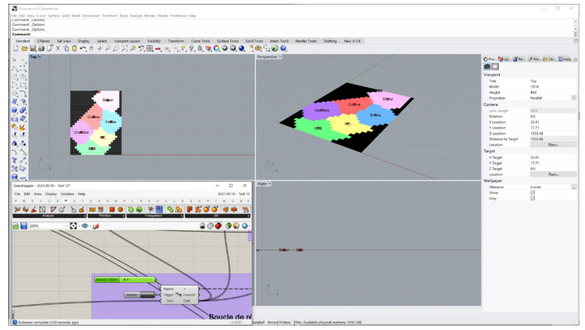


Figure 5.8.k
Itérations générées avec le script Python
© Arthur ROULAND

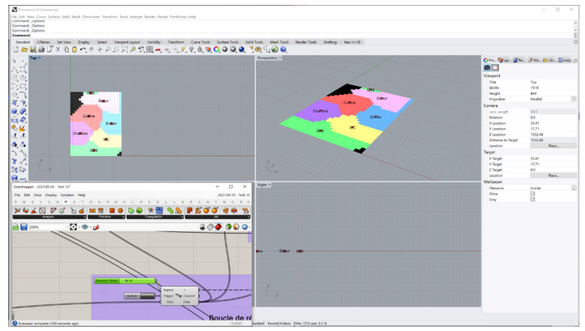


Figure 5.8.l
Itérations générées avec le script Python
© Arthur ROULAND

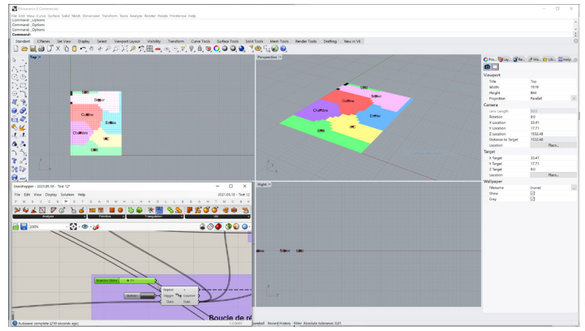


Figure 5.8.m
Itérations générées avec le script Python
© Arthur ROULAND

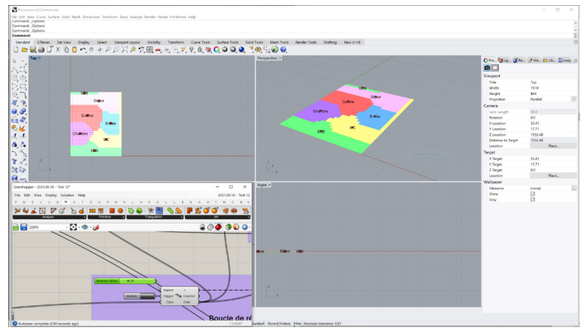


Figure 5.8.n
Résultat de la génération avec le script Python
© Arthur ROULAND

5.9 – Inclusion du code Python dans l'ensemble du programme

Afin de continuer l'élaboration de l'algorithme, j'ai densifié la définition de la grille afin d'arriver à une précision de l'ordre de la dizaine de centimètre. La surface est ainsi divisée en cellules de 10 centimètres par 10 centimètres. L'étape suivante était de récupérer ces surfaces afin de les utiliser pour former des pièces. Les formes générées sont biscornues, peu praticables et peu communes. Afin d'arriver à un résultat plus "classique" j'ai décidé de rajouter une partie dans le programme me permettant d'inscrire les surfaces générées dans des rectangles pour simplifier la partition du logement. Cependant, en augmentant la précision de la grille, j'ai également augmenté le temps de calcul de mon programme qui pouvait parfois mettre plusieurs minutes pour générer mes taches d'encre.

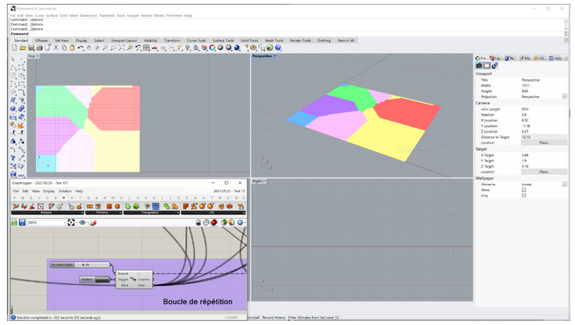


Figure 5.9.a
Image de la grille avec une définition de 10 cm par 10 cm
© Arthur ROULAND

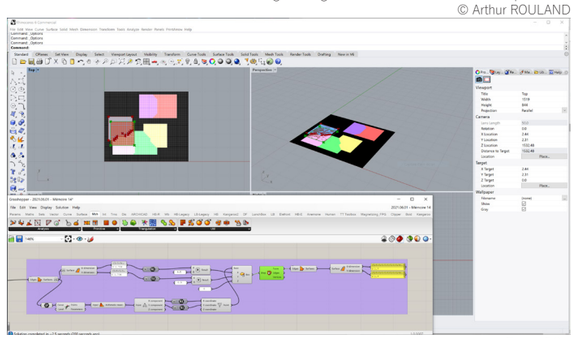


Figure 5.9.b
Problème d'inscription de la forme dans un rectangle
© Arthur ROULAND

J'ai également commencé à m'intéresser à la manière dont les premières valeurs allaient être disposées sur la grille. L'idée était de générer des valeurs aléatoires afin que les premières gouttes des taches d'encre puissent se placer seules et aléatoirement. A ce stade de la recherche, la manière de disposer ces cellules de départ aléatoirement n'avait pas encore été trouvée tout comme l'inscription des formes dans des rectangles.

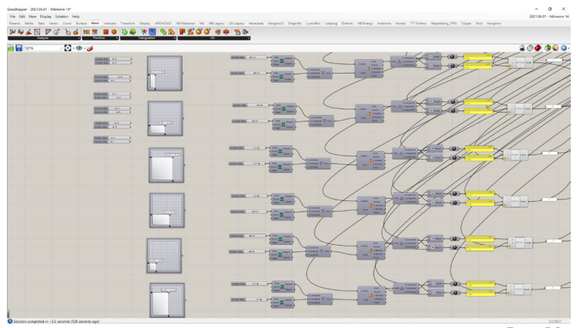


Figure 5.9.c
Tentative de génération de valeurs aléatoires
© Arthur ROULAND

5.10 – Itération régulières et enregistrement

La solution à laquelle j'ai pensé pour palier à ce problème était la création d'un timer (minuteur). Pour cela j'avais besoin de deux choses. Un composant me permettant de générer des valeurs aléatoires et un second composant pouvant activer le premier à intervalles réguliers. Ainsi, j'ai utilisé le composant "Timer" associé au composant "Deconstruct Date" afin de pouvoir envoyer un signal "True" au composant "Random". Le timer est réglé sur 120 000 secondes (2 minutes) car cela correspond à peu près au temps de génération d'un plan. Toutes les 2 minutes, cet élément active le composant "Random". Ce dernier avait comme paramètres d'entrée 12 pour le nombre de valeurs à générer (2 par pièces, une valeur en X et une valeur en Y) et un domaine dans lequel devaient se trouver les valeurs (entre 0.00 et 1.00). En effet, les dimensions de la surface générale ont été "reparamétrisées". Cela signifie que quelque soit la taille du plan, l'algorithme va appréhender la surface comme étant un axe dont 0 est le début et 1 en est la fin. Cela permet de pouvoir rentrer les valeurs que l'on souhaite pour la taille du plan sans avoir à manipuler les coordonnées de départ des pièces. Avec cet algorithme, 12 valeurs aléatoires correspondant à 6 coordonnées sur le plan étaient générées toutes les 2 minutes.

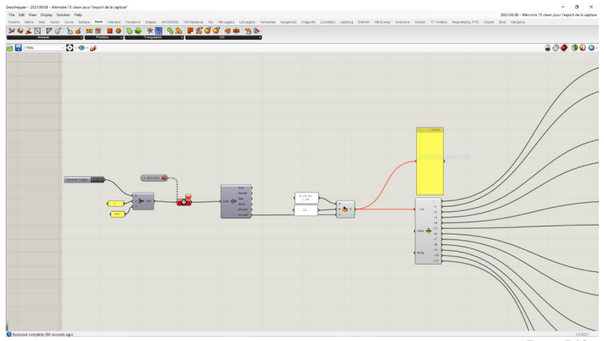


Figure 5.10.a
Timer
© Arthur ROULAND

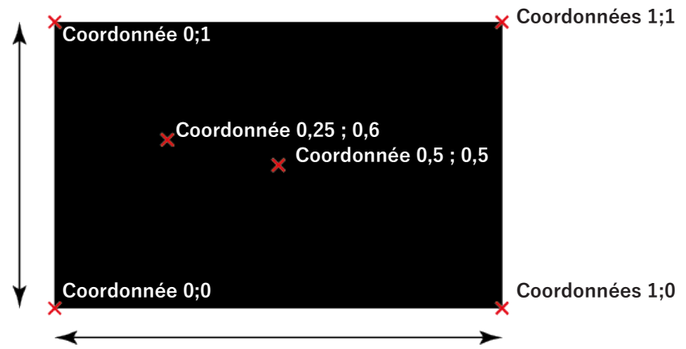


Figure 5.10.b
Illustration du «reparamétrage» d'une surface
© Arthur ROULAND

A ce stade, je me suis également penché sur les opérations entre les surfaces. En effet, nous avons vu dans l'étape précédente que les formes générées avaient pour vocation d'être inscrites dans des rectangles. Malgré le fait que cette étape ne fonctionnait pas parfaitement, il était possible de commencer à régler les opérations qui allaient s'opérer entre ces éléments. Pour cela j'ai créé une partie de l'algorithme permettant d'effectuer des opérations booléennes. Ces opérations consistent à supprimer une partie d'une surface lorsqu'elle se superpose avec une autre. Il est important de noter que cette démarche implique un ordre de priorité. Ainsi les pièces ont été connectées afin que le séjour soit la pièce la plus importante, puis la chambre, puis la cuisine etc...

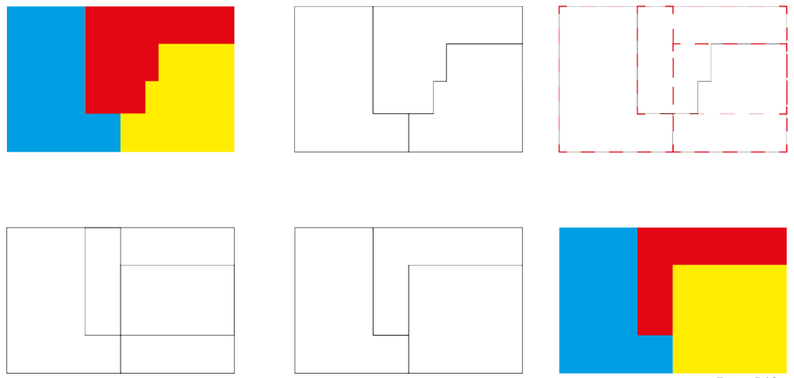


Figure 5.10.c
Inscription des surfaces dans des rectangles et opérations booléennes
© Arthur ROULAND

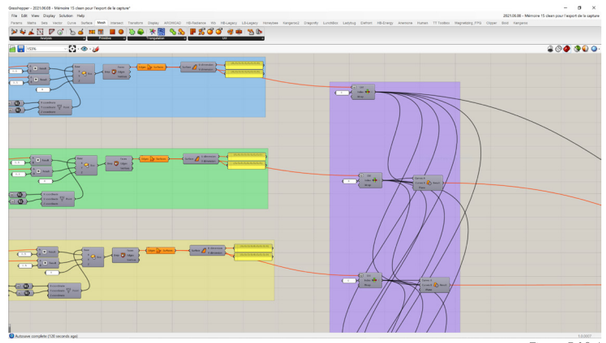


Figure 5.10.d
Programme d'opération booléenne
© Arthur ROULAND

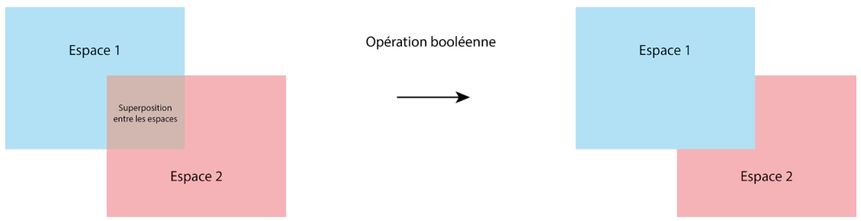


Figure 5.10.e
Schéma d'une opération booléenne
© Arthur ROULAND

Pour terminer, j'ai également orienté mes recherches sur la façon d'enregistrer les plans générés à chaque itération. Je me suis tout d'abord basé sur le mémoire de Jean-Raphael PIQUARD qui avait lui aussi eu besoin d'enregistrer ses itérations. Malheureusement, pour une raison inconnue, cela ne fonctionnait pas. J'ai donc essayé diverses méthodes afin d'enregistrer mes plans. A cette étape, j'étais en mesure d'enregistrer les plans au format .dwg (drawing). Ce format est très utilisé par les architectes et d'autres corps de métier pour travailler. Ce format enregistre des formes, des couleurs et des attributs propres aux formes dans un format qu'il est possible d'ouvrir pour le modifier. Il était nécessaire pour cette expérimentation d'enregistrer les plans au format image (.jpg, .png, .tiff etc⋯). L'enregistrement au format .dwg n'était donc pas la solution appropriée. Cependant, cela m'a permis de commencer à élaborer un algorithme d'enregistrement et de pouvoir vérifier qu'il fonctionnait correctement.

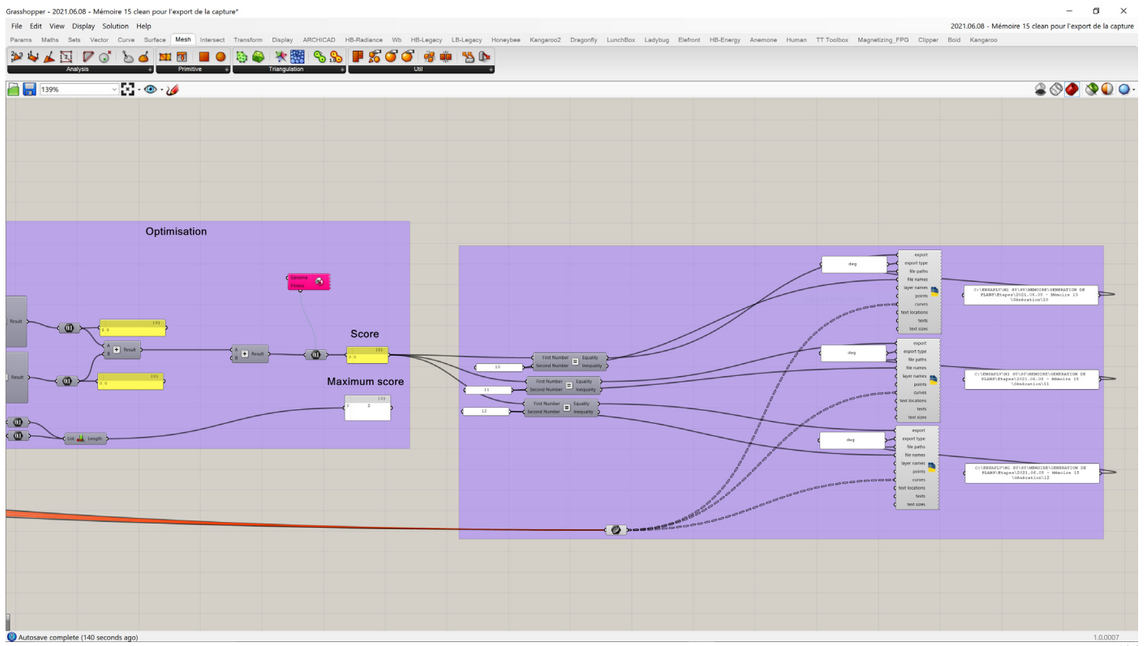


Figure 5.10.f
Premier programme d'enregistrement des itérations
© Arthur ROULAND

Arthur ROULAND

5.11 – Résolution des problèmes du composant Python

Afin de solutionner le problème lié au programme Python, j'ai appelé une seconde fois mon ami Ewen COSSEC. Nous avons finalement réussi à solutionner le problème de passage des couleurs d'un côté à l'autre. Cependant, mes recherches ayant avancé, je lui ai fait part de mes intentions afin d'améliorer le programme. En effet, jusqu'à présent, il était nécessaire d'indiquer à l'algorithme le nombre d'itérations souhaité pour générer un plan. Ce nombre d'itérations était variable en fonction du positionnement des cellules de départ et de la taille du plan. J'avais réglé par défaut un nombre surévalué. Cela affectait les performances de mon ordinateur et donc le temps nécessaire à la génération d'un plan. De plus, jusqu'à présent, j'avais besoin de visualiser l'état de propagation des couleurs sur la grille. Cela me permettait de tester mes scripts Python afin de m'assurer que le programme fonctionnait correctement. Le problème étant réglé, je n'avais plus besoin de visualiser chaque itération. Cela était également souhaité car l'algorithme envoyait en sortie chaque étape de la répartition des couleurs. De ce fait, chaque étape était alors ensuite envoyée dans la création de rectangles, dans le système d'opération, dans le système d'évaluation etc ⋯ Cela ralentissait considérablement mon ordinateur alors que je n'avais pas besoin de ces évaluations intermédiaires. Nous avons donc travaillé sur une amélioration du programme qui allait nous permettre d'effectuer toutes les itérations dans le script Python pour n'envoyer en sortie que le résultat final une fois la grille entièrement colorée. Grâce à ce changement, le temps de génération d'un plan passait de 120 000 secondes (2 minutes) à moins de 20 secondes. Le voisinage étant déjà paramétré dans le script Python, deux étapes présentes dans GrassHopper n'étaient plus d'aucune utilité ce qui a largement participé à la simplification du programme. L'utilisation de l'add-on "Anemone" a donc été arrêtée à cette étape car le programme pouvait désormais savoir s' il était nécessaire de continuer ou non sans avoir recours à Anemone.

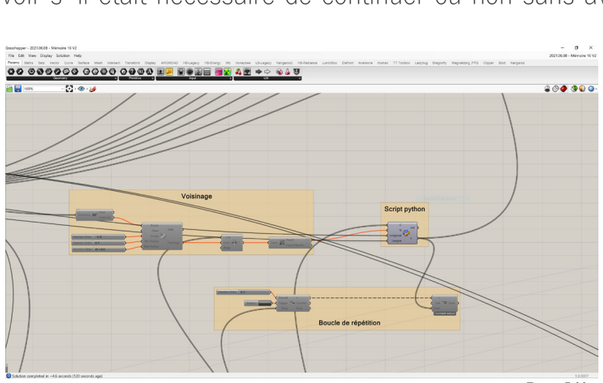


Figure 5.11.a
Désactivation de l'add-on Anemone
© Arthur ROULAND

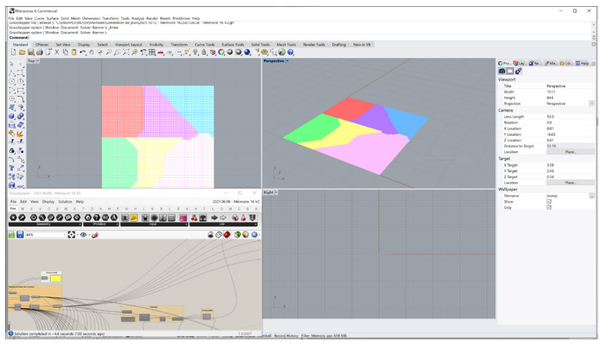


Figure 5.11.b
Résolution du problème de propagation des couleurs
© Arthur ROULAND

5.12 - Premiers enregistrements

Suite à cela, j'ai pu continuer mes recherches concernant l'enregistrement des plans. Je me suis rendu sur le site Rhinoceros Forums sur lequel j'ai pu récupérer un composant "Script C#" qu'un dénommé David Rutten avait mis en ligne.

https://discourse.mcneel.com/t/capturing-rhino-layout-viewport-iterations-print-or-image-export/51387

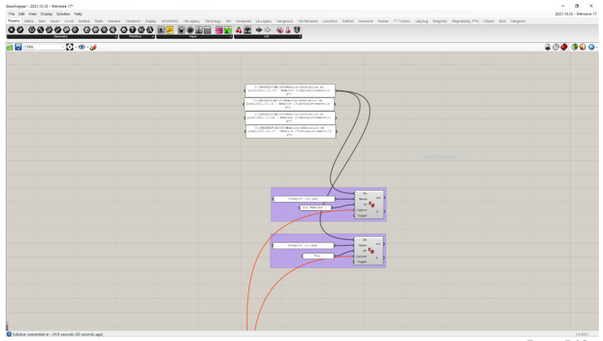


Figure 5.12.a
Script C# pour l'enregistrement
© Arthur ROULAND

Ce composant était exactement ce que je cherchais. Il m'a donc permis d'enregistrer chaque itération dans un dossier spécifique, au format .png en donnant un nom à chaque image. J'avais pu essayer d'enregistrer différemment au cours de ma recherche notamment à l'aide de l'add-on LadyBug. J'ai pu enregistrer quelques images mais cette méthode présentait un défaut très important. L'idée de ce programme était d'enregistrer les plans, de 6 points par exemple, de la manière suivante :

Sa destination : L'enregistrer dans le dossier de sa typologie (par exemple T2)

Sa vue : Une vue a été configurée dans Rhinoceros afin de paramétrer le cadrage d'enregistrement du plan

Son nom : Plan 6 pts {0}.png

Le nom du fichier est très important. En effet, en utilisant LadyBug, les plans s'enregistrent les uns sur les autres. C'est-à- dire que le nouveau plan à 6 points supprimait l'ancien. En utilisant le script C# de David Rutten, le "{0}" permettait à l'ordinateur de donner un numéro au plan. Ainsi le premier plan enregistré s'appellera "Plan 6 points 0.png" et le suivant "Plan 6 points 1 .png". Cela m'a donc permis de catégoriser les plans en leur donnant un nom identifiable mais unique afin de pouvoir tous les enregistrer. J'ai tout de même eu quelques imprévus. Après avoir modifié certaines parties de mon programme, celui-ci n'enregistrait que des images vides ou incomplètes. Le problème était que le composant Script C# doit être le composant le plus en avant dans l'espace GrassHopper. Ce problème a été réglé en copiant et collant simplement le composant afin de le placer en avant dans le plan de travail de GrassHopper.

Figure 5.12.b
Logo de l'add-on LadyBug
Source : food4rhino.com

En parallèle, j'ai également remarqué que le composant "Random" qui me servait à générer des valeurs aléatoires comprises entre 0.00 et 1.00 ne fonctionnait pas véritablement comme je le souhaitais. En effet, au bout de quelques générations, j'ai noté que les valeurs étaient identiques à celles des premières générations. L'aspect aléatoire des valeurs est très important dans cette recherche afin d'obtenir une diversité infinie de plans. Le fait que les générations se répètent va engendrer des générations de plans identiques et cela n'est pas souhaitable. J'ai donc cherché un moyen d'obtenir réellement des valeurs aléatoires pour mon programme. Je me suis rendu sur le site Rhinoceros Forums et j'ai pu tester un programme intitulé "Randoms_Unique_viaHashSet_V1.gh" posté sur le forum par un certain Peter Fotiadis.

https://discourse.mcneel.com/t/real-random-numbers/56072/5

J'ai adapté le programme contenu dans ce fichier et après plusieurs tests afin de vérifier le caractère aléatoire des valeurs j'ai décidé de le conserver et de l'utiliser dans le cadre de ma recherche. Grâce à ces programmes, j'étais alors en mesure de générer aléatoirement un plan toutes les 20 secondes et de l'enregistrer correctement sur mon disque dur.

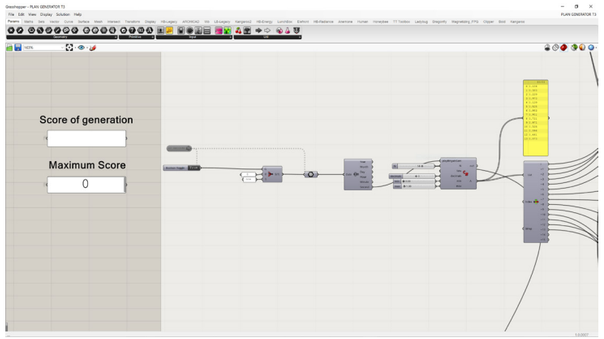


Figure 5.12.c
Ajout du script C# dans le timer
© Arthur ROULAND

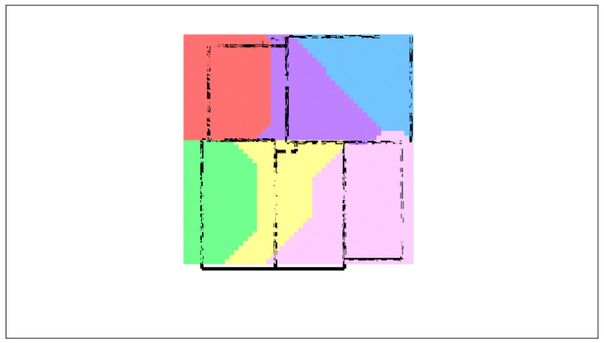


Figure 5.12.d
Premiers enregistrements
© Arthur ROULAND

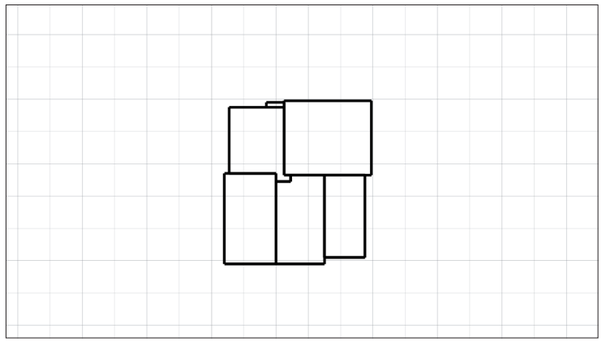


Figure 5.12.e
Premiers enregistrements
© Arthur ROULAND

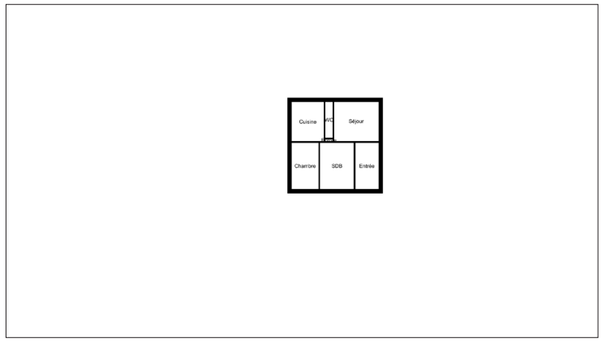


Figure 5.12.f
Premiers enregistrements
© Arthur ROULAND

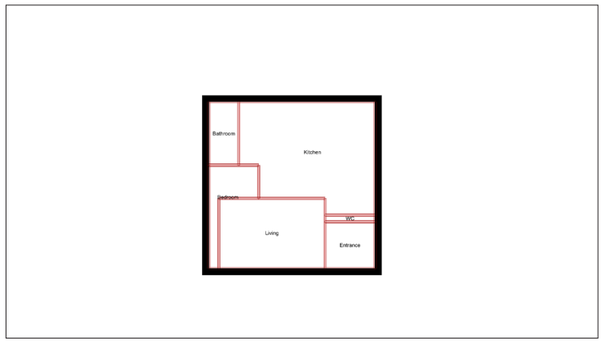
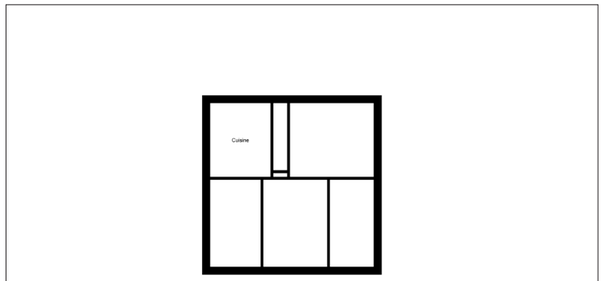


Figure 5.12.g
Premiers enregistrements
© Arthur ROULAND



5.12.h
Premiers enregistrements
© Arthur ROULAND

6

Résultats de la recherche

Arthur ROULAND

6.0.1 - Vue d'ensemble du programme final

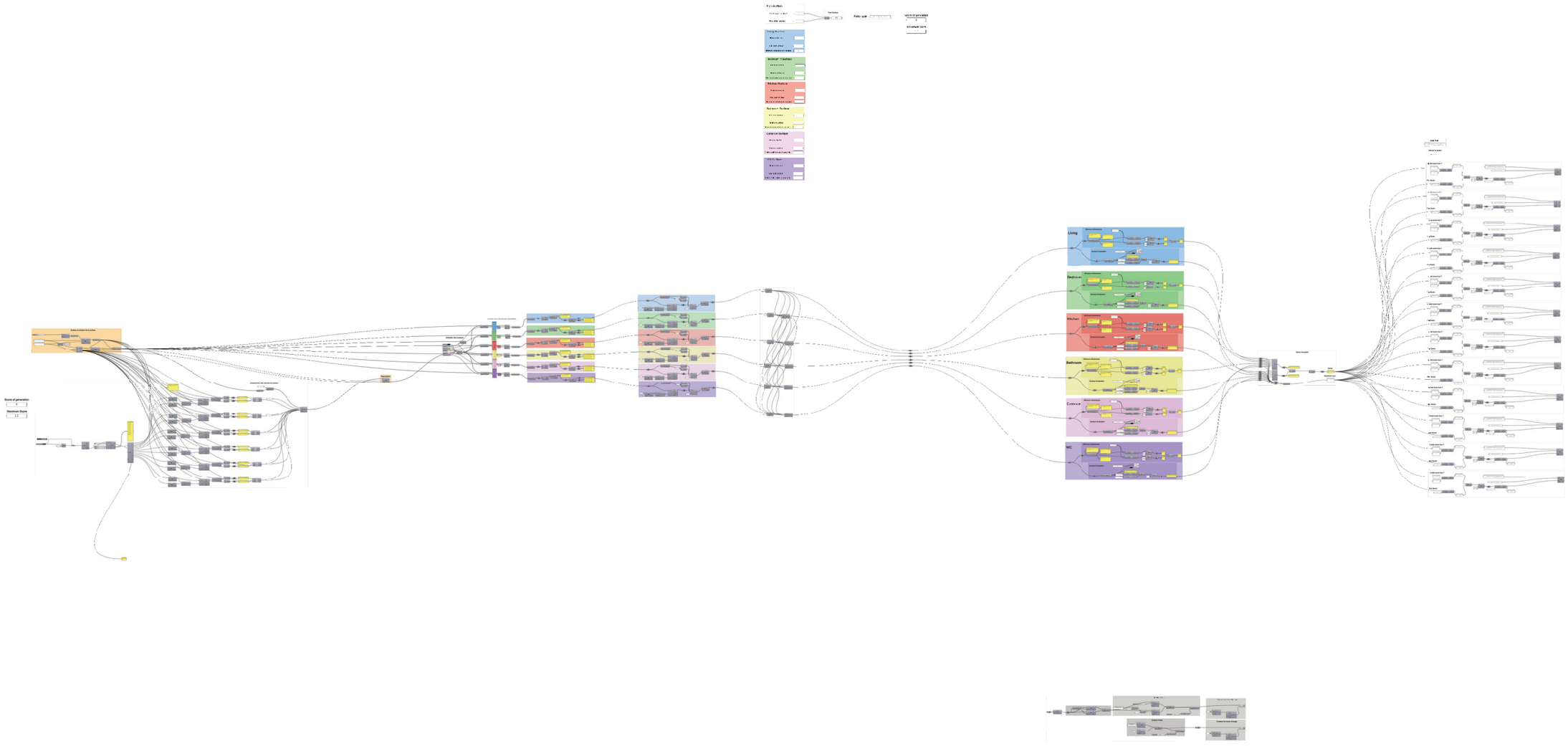


Figure 6.0.1.a
Vue d'ensemble du programme final
© Arthur ROULAND

Arthur ROULAND

6.0.2 - Présentation générale de l'algorithme

L'algorithme final se compose en 12 parties ayant chacune une fonction spécifique.

- La partie 1 propose à l'utilisateur d'agir sur certains paramètres

- La partie 2 sert à définir la surface et à la diviser

- La partie 3 permet de générer des valeurs aléatoires à intervalle régulier.

- La partie 4 est le script Python qui va nous permettre de diviser la surface en plusieurs zones distinctes.

- La partie 5 attribue à chaque zone une couleur différente afin de pouvoir les identifier.

- La partie 6 consiste en la simplification des surfaces générées en les inscrivant chacune dans un rectangle.

- La partie 7 réalise des opérations booléennes entre les différentes surfaces afin qu'aucune d'entre elles ne se chevauchent.

- La partie 8 est la récupération des contours des surfaces finales

- La partie 9 permet de regarder si les surfaces générées et découpées répondent aux paramètres de départ et leur attribuent des scores.

- La partie 10 évalue le score total du plan et le score maximal qu'il aurait pu obtenir.

- La partie 11 récupère les périmètres des différents espaces et crée des lignes noires épaisses symbolisant des cloisons intérieures et des murs extérieurs.

- La partie 12 permet l'enregistrement des plans dans un dossier sur le disque de l'ordinateur.

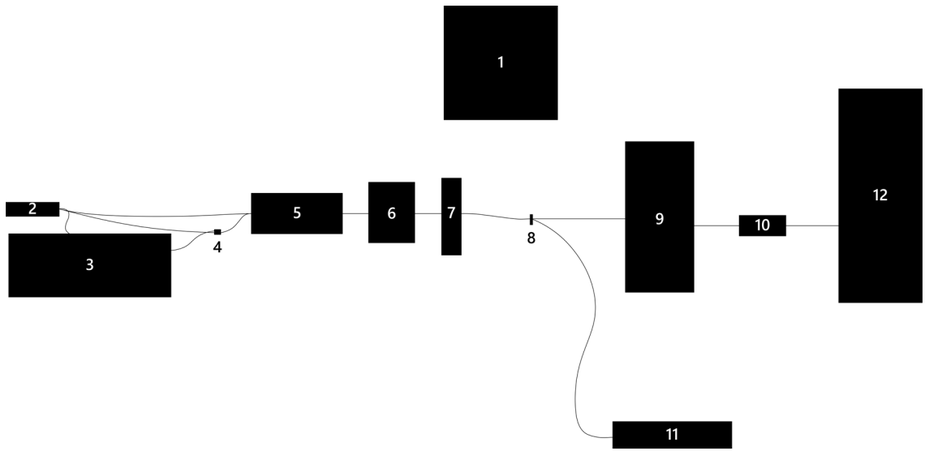


Figure 6.0.2.a
Schéma des parties du programme final
© Arthur ROULAND

6.1 - Paramètres de départ

La première partie de ce programme permet de définir les paramètres de départ sur lesquels nous pouvons agir. Dans un premier temps, nous pouvons indiquer la longueur et la largeur du plan en entrant une valeur en mètre dans les panneaux blancs. Ces deux valeurs sont ensuite multipliées afin d'afficher la surface totale du plan que l'on souhaite générer.

En dessous, nous retrouvons 6 groupes, chacun d'une couleur différente. Ce programme étant pour la génération de T2, il ne comporte que 6 espaces (séjour, chambre, cuisine, salle de bain, entrée et WC). Dans le programme pour la génération de T3 nous retrouverons une chambre en plus et dans le programme qui génère des T4, 2 chambres en plus.

Chacun de ces groupes est donc identifié avec une couleur et le nom de l'espace correspondant en anglais afin de permettre au plus grand nombre de comprendre ce programme. Il est demandé de rentrer la surface minimale ainsi que la surface maximale souhaitée pour chacune des pièces. Il est également demandé la dimension minimale d'un côté. En effet, sans ce paramètre un séjour de 15 m² pourrait être une surface de 15 mètres par 1 mètre et cela n'est pas souhaitable. Ensuite, nous pouvons renseigner le chemin d'accès dans lequel nous souhaitons enregistrer nos plans. Sans cela, aucun plan ne sera enregistré.

Pour terminer nous retrouvons un panneau renseignant le score de la génération ainsi que le score maximum qu'il aurait pu atteindre.

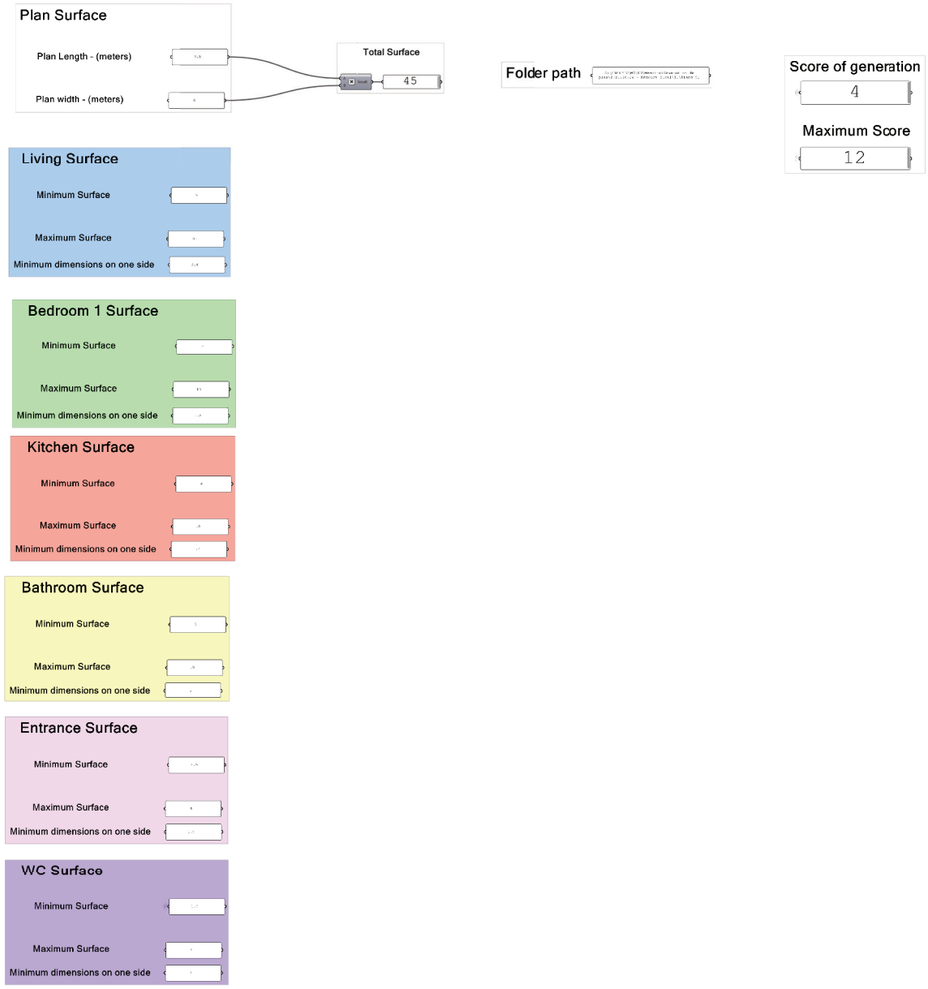


Figure 6.1.a
Partie 1 du programme final
© Arthur ROULAND

Arthur ROULAND

6.2 - Définition de la grille

La deuxième partie de cet algorithme récupère la longueur et la largeur du plan souhaité et crée une surface. Cette surface, ici de 7,5 mètres par 6 mètres, est divisée 10 fois plus précisément. On obtient alors une grille avec une précision de 10 cm. Ce paramètre peut être modifié afin d'obtenir des plans plus ou moins précis. Cela va influer sur la rapidité de la génération du plan ainsi que sur le résultat obtenu.

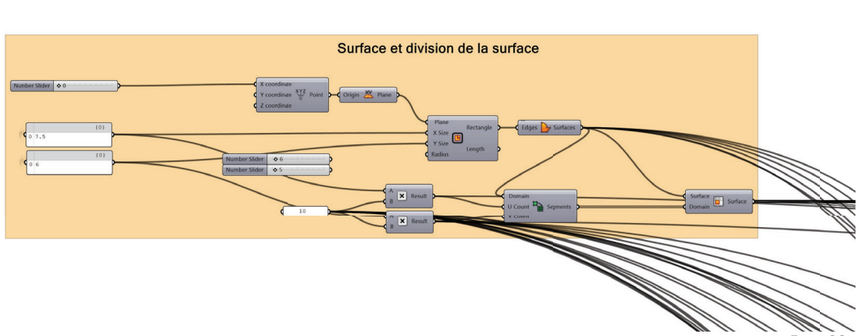


Figure 6.2.a
Partie 2 du programme final
© Arthur ROULAND

6.3 - Valeurs aléatoires de départ

La grille obtenue dans la partie 2 est récupérée dans la partie 3. Avant cela, un timer a été élaboré afin de permettre de générer des plans à intervalle régulier sans avoir à relancer le programme après chaque génération. Ici, on peut voir que le timer est réglé sur 20 secondes. Ce composant de grasshopper va activer toutes les 20 secondes le "Boolean Toggle". Ce dernier va envoyer un signal "True" au composant "Stream Filter". Ce composant va ordonner aux autres éléments de s'activer. Le timer est à adapter à chaque ordinateur. Si l'ordinateur est puissant quelques dizaines de secondes suffisent. Si l'ordinateur n'a pas des capacités de calcul très élevées, alors un temps plus long sera nécessaire.

On retrouve ensuite un code avec des sliders. Ce composant de code va nous permettre de générer des valeurs aléatoires. Les sliders permettent de paramétrer le nombre de valeurs qui vont en sortir. Pour ce programme générant des T2 de 6 pièces, nous avons besoin de deux fois six valeurs. C'est pourquoi le slider "N" (pour number) est réglé à 12. Les autres sliders indiquent le domaine dans lequel vont se trouver ces valeurs aléatoires. Ici, on voit que l'on va générer 12 valeurs, comprises entre 0 et 1 avec au maximum 3 décimales.

Enfin, les données générées sont affichées dans un composant "Panel" en jaune. Dans cet exemple, 12 valeurs sont générées, numérotées de 0 à 11. Elles ne sont ni logiques, ni rangées dans quelconque ordre. En dessous, nous retrouvons le composant "List Item". Cet élément permet de récupérer une liste et d'en sortir certains éléments. On voit en sortie que chaque valeur aléatoire est utilisée séparément des autres afin d'être envoyée dans la suite de cette partie de l'algorithme.

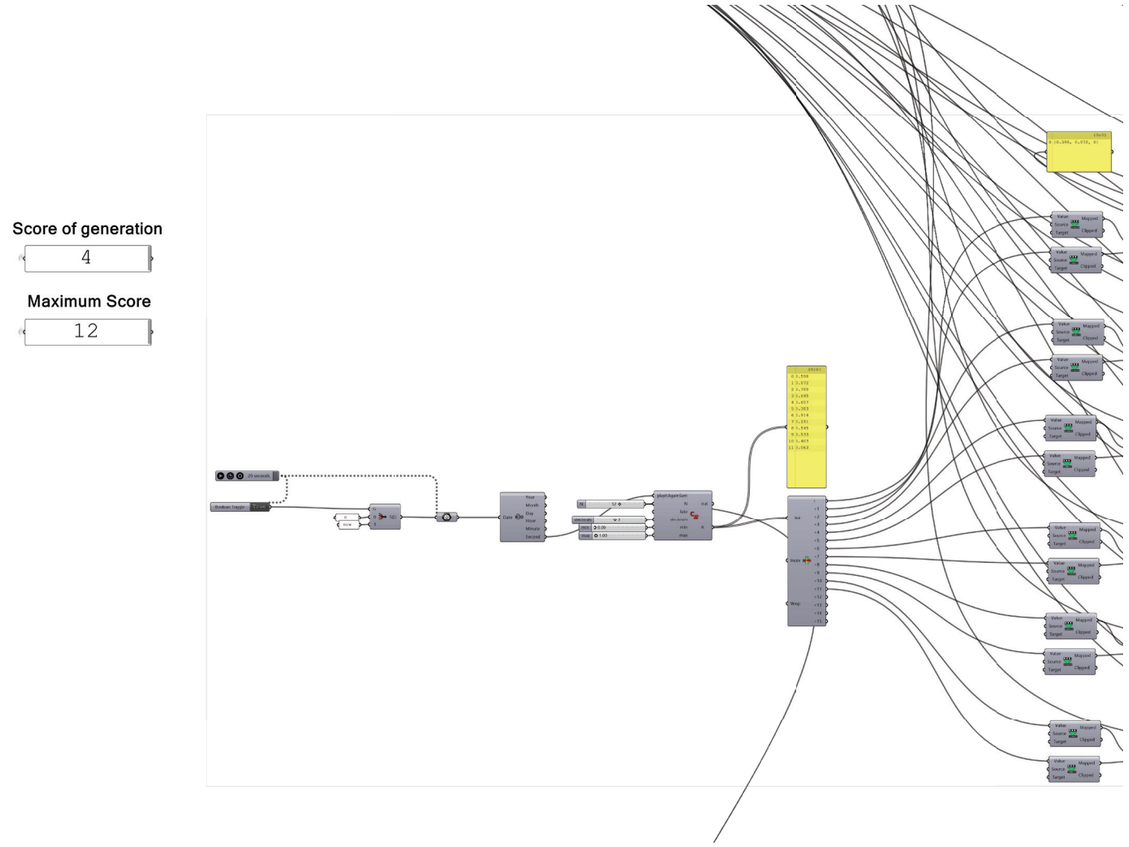


Figure 6.3.a
Partie 3 du programme final : Timer et valeurs aléatoires
© Arthur ROULAND

Après la génération de 12 valeurs aléatoires, l'algorithme se divise en 6 parties identiques. Chacune de ces parties va donc correspondre à une pièce. Pour chaque pièce, nous avons besoin d'une valeur de départ (cela sera expliqué plus précisément dans la partie 4). Le programme récupère donc deux valeurs aléatoires générées précédemment afin de les utiliser comme coordonnées afin de placer un point au hasard sur la grille générée dans la partie 2. Chaque valeur va être disposée sur la grille. Une première valeur sur l'axe X est comprise entre 0 et 1. 0 étant le point le plus à gauche du plan et 1 le point le plus à droite du plan (dans cet exemple 7,5 mètres). Une seconde valeur sur l'axe Y entre 0 et 1 (ici entre 0 et 6 mètres). Ce point est donc disposé sur la surface et est associé à une cellule du plan. Cette valeur a été multipliée par la longueur du plan sur ce même axe. On obtient 2 valeurs. Dans cet exemple de T2 de 7,5 mètres par 6 mètres, les coordonnées possibles sont comprises en X entre 0 et 75 et en Y entre 0 et 60.

Ces deux valeurs sont ensuites utilisées dans un composant "Expression" qui contient la formule suivant : L x j + i
i et j sont ici les deux valeurs des coordonnées finales. Cependant, il est nécessaire d'expliquer que la grille est composée de cellules (ici 75 x 60 = 4 500 cellules) numérotées de 0 à 4 499. Cependant, cela forme une ligne et nous souhaitons paramétrer ces cellules afin qu'elles forment un rectangle. Ce composant nous permet de dire que notre grille sera composée de 75 colonnes de chacune 60 cellules. Nous obtenons donc pour la première pièce de ce plan, des coordonnées indiquant la colonne 45 et la ligne 52. A cet endroit se trouve une cellule et cette dernière se voit attribuer la valeur 1 par le composant "List Item". Cela s'applique aussi à chacune des 5 autres pièces ayant chacune des coordonnées aléatoires en X et en Y afin de définir la première cellule de chaque espace.

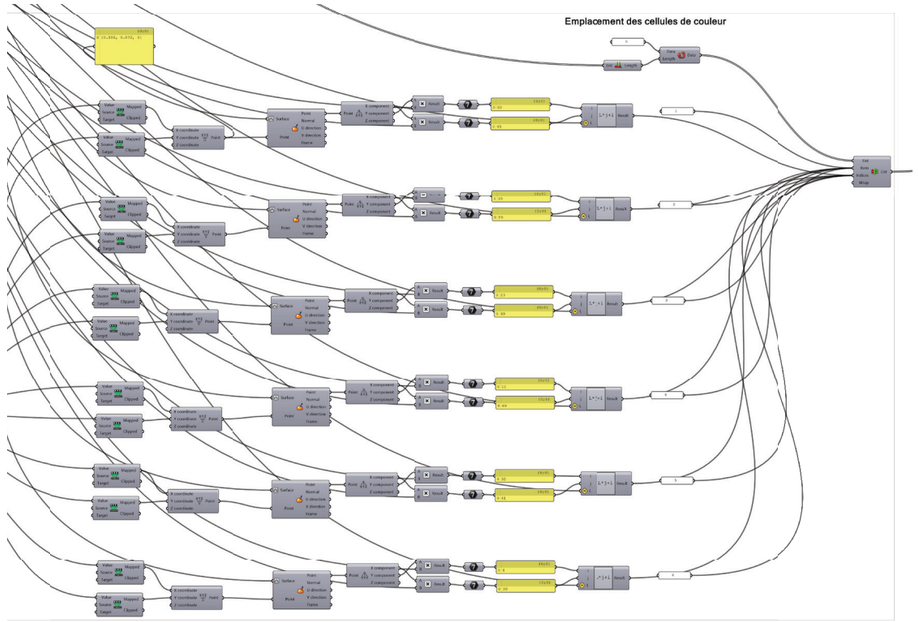


Figure 6.3.b
Partie 3 du programme final : Activation des cellules initiales
© Arthur ROULAND

6.4 - Division de la grille

La quatrième partie est composée d'un seul élément : un script python. Comme nous avons pu le voir précédemment dans la partie 5 de ce mémoire, la division "aléatoire" du plan était un réel enjeu afin d'obtenir des résultats différents à chaque itération. Pour le programme final, j'ai opté pour un système que je définis comme un système en taches d'encre. Imaginons une feuille blanche imbibée d'eau sur laquelle nous venons déposer 6 gouttes d'encre de différentes couleurs. Ces gouttes vont s'étaler sur cette feuille, se rencontrer afin que la feuille soit entièrement recouverte de couleurs. Ce script python a pour rôle d'utiliser nos cellules initiales afin de les faire s'étaler de cette même manière. Le script python est disponible et détaillé dans son intégralité dans la partie "Documents annexes".

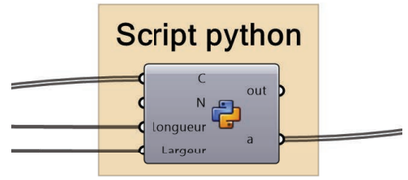


Figure 6.4.a
Partie 4 du programme final
© Arthur ROULAND

Le script utilise 3 entrées différentes. Une entrée C dans laquelle est renseignée les coordonnées de départ de chaque cellule initiale avec sa valeur. Une valeur correspond à une pièce (qui va par la suite être visualisée par une couleur). On retrouve également une entrée "Longueur" et une entrée "Largeur". Ces entrées sont reliées à la multiplication de la longueur et de la largeur par 10. Pour le plan de 7,5 mètres par 6 mètres on a comme paramètre de longueur la valeur 75 et pour le paramètre de largeur, 60. La grille est organisée de la manière suivante.

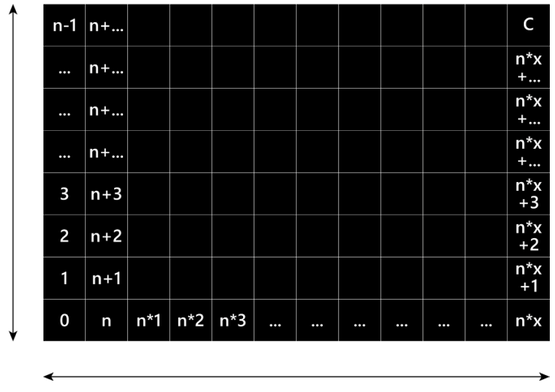


Figure 6.4.b
Schéma d'organisation de la grille
© Arthur ROULAND

Arthur ROULAND

Cette grille est donc organisée en cellules ayant pour la plupart 8 voisins : 3 au-dessus, 3 en dessous et 1 de chaque côté, ce qui correspond au voisinage de Moore expliqué précédemment dans la partie 5.8. Afin de faire fonctionner ce programme il a fallu identifier les cellules qui ne correspondaient pas à ces cellules dites "standards". Il s'agit donc de toutes les cellules se trouvant sur les bordures de notre grille. Pour ce faire, j'ai créé quatre paramètres correspondant chacun à une des bordures de la grille : LigneDuHaut, LigneDuBas, ColonneGauche et ColonneDroite. Chacun de ces composants contient les voisins à exclure pour que le programme fonctionne correctement. Exception dans l'exception, les cellules situées dans les coins de la grille doivent exclure plus de voisins que les cellules qui sont simplement en bordure de la grille.

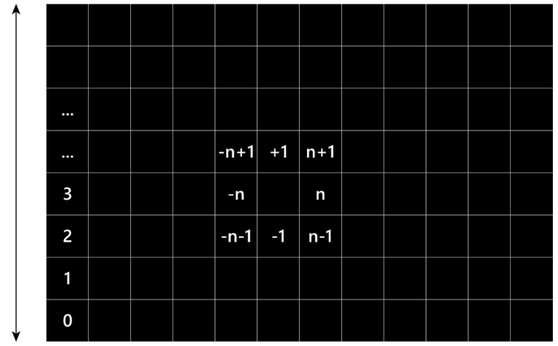


Figure 6.4.c
Etude des voisins d'une cellule
© Arthur ROULAND

Notre automate cellulaire fonctionne en observant les voisins de chaque cellule. On souhaite que toutes les cellules soient à l'état 0 sauf 6 d'entre elles ayant pour valeur un numéro entre 1 et 6 chacune correspondant à une couleur. A chaque itération du script python, les cellules à l'état 0 vont regarder autour d'elles si un de leur voisin a une valeur (et donc une couleur). Si elle n'ont pas de voisin de couleur, elles restent à l'état 0 (noir). Si un de leur voisin a une couleur (par exemple vert, correspondant à la valeur 2), alors la cellule va s'attribuer la valeur 2 et devenir verte. Cependant, notre programme étudie les cellules de la cellule 0 à la cellule C (C correspondant à la longueur totale de la grille, soit, la dernière cellule). Si une cellule devient verte, lorsque la prochaine cellule va regarder ses voisins, elle verra une cellule verte et donc prendra également sa couleur. Cela fausse donc le résultat. Pour ce faire, notre algorithme dit à chaque cellule que si elle a un voisin de couleur, alors elle va mémoriser cette valeur et lorsque toutes les cellules auront été étudiées, seules les cellules ayant une valeur différente de zéro pourront à ce moment là, changer d'état. Cette opération est répétée en boucle jusqu'à ce que toutes les cellules de la grille aient une couleur différente du noir (état 0).

6.5 - Attribution des couleurs

La cinquième partie consiste à identifier les différentes surfaces obtenues. Afin de mieux visualiser lors de l'élaboration de l'algorithme de génération de plan, j'ai décidé d'attribuer à chaque valeur une couleur afin de comprendre l'organisation du plan au premier coup d'œil. Ainsi, chaque valeur va se voir attribuer une couleur. La valeur 0 est noire, la valeur 1 est bleue, la 2 est verte, la 3 est rouge, la 4 est jaune, la 5 est rose et la 6 est violette. Pour les générateurs de plan de T3 et de T4, une ou deux couleurs supplémentaires ont été ajoutées correspondant à une nouvelle pièce.

Le programme se divise à nouveau en 6 parties (pour les 6 pièces du plan). Pour chaque pièce, sa couleur est utilisée afin de mieux se retrouver dans le programme. A la sortie de l'attribution des couleurs, on récupère le nombre de cellules de chaque couleur ainsi que leur numéro d'identification (compris ici entre 0 et 4 499). Chaque cellule est ensuite transformée en une surface (de 10 centimètres par 10 centimètres dans ce cas-ci). Les cellules sont ensuite assemblées entre elles afin de former la surface totale de chaque couleur. Cette surface totale est ensuite décomposée afin d'en extraire, la surface ainsi que son périmètre.

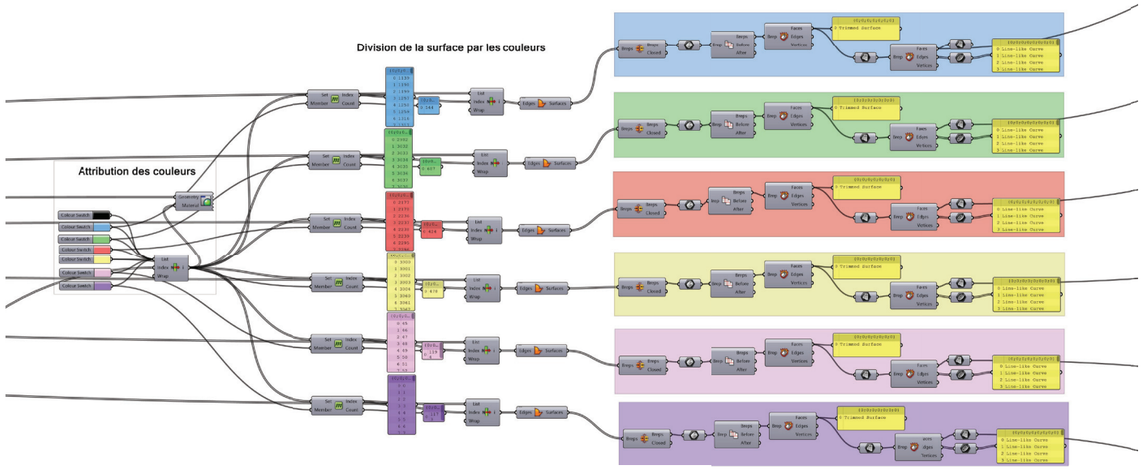


Figure 6.5.a
Partie 5 du programme final
© Arthur ROULAND

Arthur ROULAND

6.6 - Simplification des surfaces

Cette sixième partie permet d'inscrire les surfaces complexes générées dans des rectangles de mêmes dimensions afin de simplifier le plan. La surface obtenue en sortie de la cinquième partie est utilisée par le composant "Dimensions". Ce composant de GrassHopper permet d'évaluer la dimension en U et en V d'un élément. Comprenez ici en X et en Y. Ces données vont être utilisées afin de générer un rectangle avec les mêmes dimensions U et V afin d'inscrire parfaitement la surface générée dans un rectangle. Le composant utilisé afin de créer ce rectangle est le composant "Center Box". Mon choix s'est porté sur ce composant car il permet de renseigner facilement le centre d'une boîte. Par défaut, chaque boîte est construite comme ayant pour centre le point X=0 et Y=0. Le composant "Center Box" crée des boîtes à partir de son centre. Si l'on rentre la valeur 2 en X et 3 en Y, ce composant va créer un objet de 2 unités en -X, de deux unités en X, de 3 unités en -Y et de 3 unités en Y. Au lieu d'obtenir un rectangle de 6 m² de surface, on obtient un rectangle de 24 m² de surface. Afin d'y remédier, les valeurs en U et en V sont multipliées par 0.5 afin d'avoir un rectangle ayant les dimensions souhaitées. La valeur Z est de 0 car nous cherchons à obtenir une surface et non pas un volume.

Afin de récupérer le centre de chaque surface, les dimensions en X et en Y vont être étudiées comme allant de 0 à 1. Ainsi avec la valeur 0.5 en X et en Y, on obtient le centre de la surface complexe générée et on utilise ce même point comme centre de référence de la surface rectangle.

En sortie de cette partie numéro 6, on obtient donc 6 surfaces rectangulaires s'inscrivant dans le plan mais se superposant.

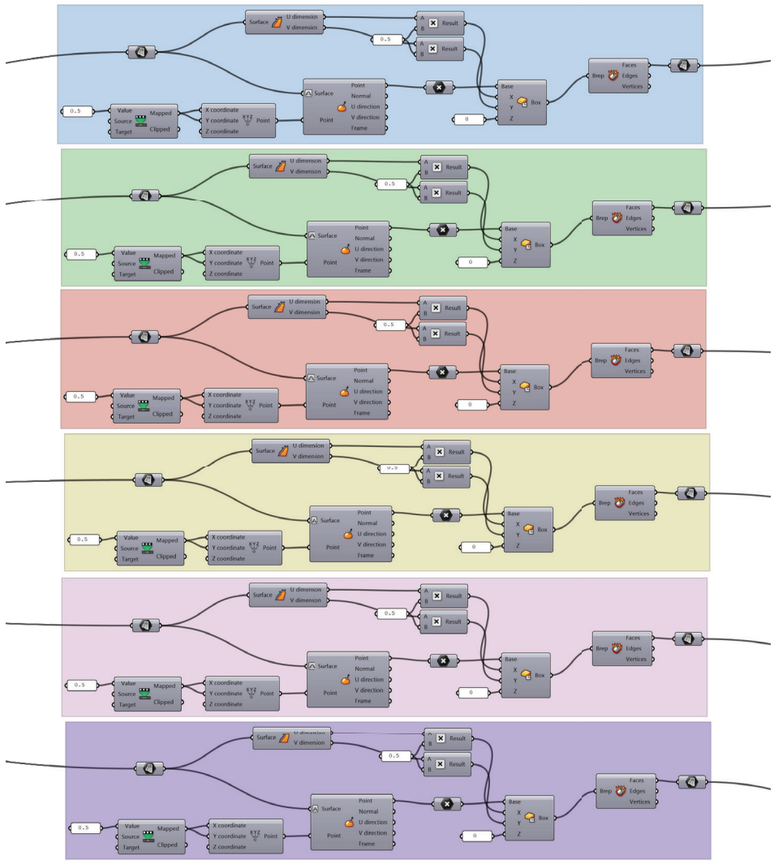


Figure 6.6.A
Partie 6 du programme final
© Arthur ROULAND

6.7 - Opérations entre les surfaces

La septième partie va nous permettre de récupérer les surfaces générées dans la sixième partie et de les découper afin qu'aucune d'entre elles ne se superpose. Pour ce faire j'utilise le composant "Region Difference" de GrassHopper. Ce composant permet de créer des opérations booléennes entre les surfaces. Si deux surfaces se superposent, alors l'une d'entre elles est prioritaire sur l'autre. La seconde surface va donc être coupée par la première comme on peut le voir sur le schéma ci-dessous.

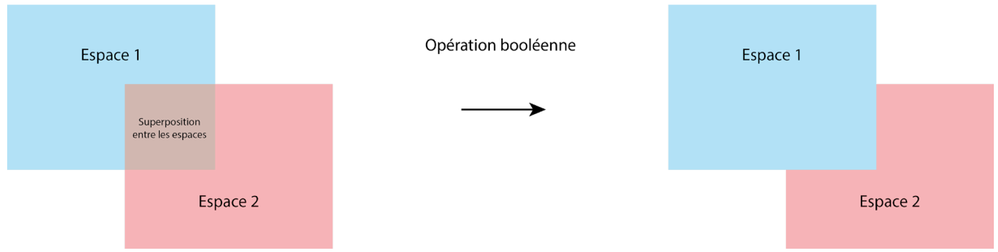


Figure 6.7.a
Schéma de principe d'une opération booléenne
© Arthur ROULAND

Cette opération implique un ordre de priorité de certains espaces sur les autres. Dans ce programme, l'espace du séjour (Living) est prioritaire sur toutes les autres surfaces. Ensuite, la chambre (Bedroom) est prioritaire sur tous les espaces sauf sur celui du séjour. Puis vient le tour de la cuisine (Kitchen), puis de la salle de bain (Bathroom), de l'entrée (Entrance) et enfin des sanitaires (WC). Dans le programme de génération de plan de T3 et de T4, une à deux chambres sont ajoutées entre la chambre 1 et la cuisine. Ainsi la chambre 1 est prioritaire sur la chambre 2 qui elle même est prioritaire sur la chambre 3 (pour les plans de T4). Ainsi, on obtient en sortie 6 surfaces qui ne se superposent plus et qui sont découpées afin que l'entièreté du plan soit utilisée.

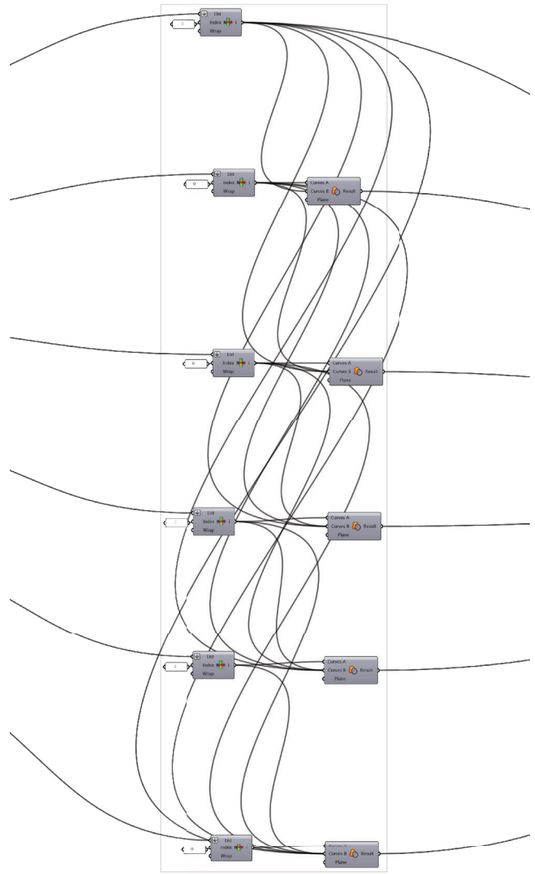


Figure 6.7.b
Partie 7 du programme final
© Arthur ROULAND

Arthur ROULAND

6.8 - Récupération des contours

La partie numéro 8 consiste uniquement à récupérer les contours des surfaces finales afin de pouvoir les visualiser facilement si besoin. Elles sont ensuite envoyées dans les parties 9 et 11.

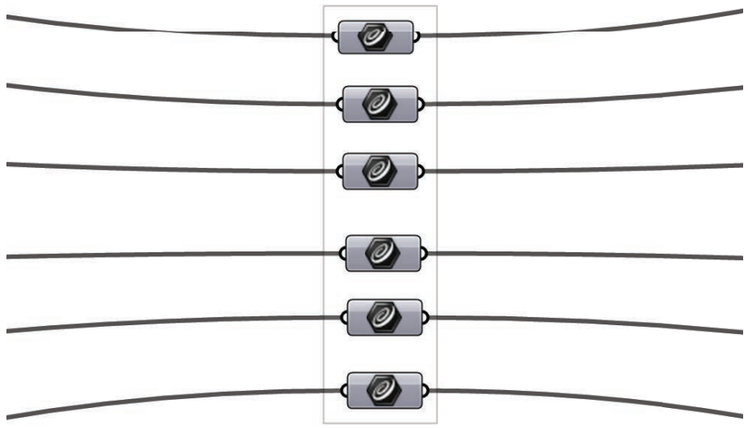


Figure 6.8.a
Partie 8 du programme final
© Arthur ROULAND

6.9 - Evaluation des scores

La neuvième partie permet d'évaluer le score de chaque espace. Chaque surface est récupérée et utilisée dans 2 algorithmes.

Le premier permet de vérifier si les dimensions de chaque espace correspondent aux valeurs rentrées dans la partie 1. A l'aide du composant "Dimensions" nous obtenons les valeurs en U et en V (en X et en Y) de chaque surface. Ces valeurs sont comparées à la valeur renseignée dans la partie 1 à l'aide du composant "Larger Than". Si la valeur par exemple en X est supérieure ou égale à la valeur minimale souhaitée alors le composant "Stream Filter" va générer la valeur 1. Si elle est inférieure, ce composant va générer la valeur 0. Cette opération est appliquée à la longueur et à la largeur de la surface. Ces deux valeurs sont ensuite multipliées entre elles afin d'obtenir un score de 1 point ou de 0 point. Ainsi chaque surface obtient 1 point pour son respect de la dimension minimale de la pièce.

Dans un second temps, la surface est elle aussi évaluée. Le composant "Area" permet de connaître la surface d'un objet complexe. On utilise ici les composants "Larger Than" et "Smaller Than" afin d'évaluer si notre surface est comprise dans les valeurs souhaitées. La surface maximale du salon est volontairement grande afin que cette pièce puisse être la plus grande possible. Les composants "Larger Than" et "Smaller Than" se rejoignent ensuite dans l'élément "Gate And". Cela permet de réunir les deux informations. Si les deux évaluations sont positives, alors le composant "Stream Filter" va envoyer la valeur 1. Si l'une des deux évaluations s'avère être mauvaise, alors il enverra la valeur 0. Ainsi chaque surface obtient 1 point pour son respect de la surface de la pièce.

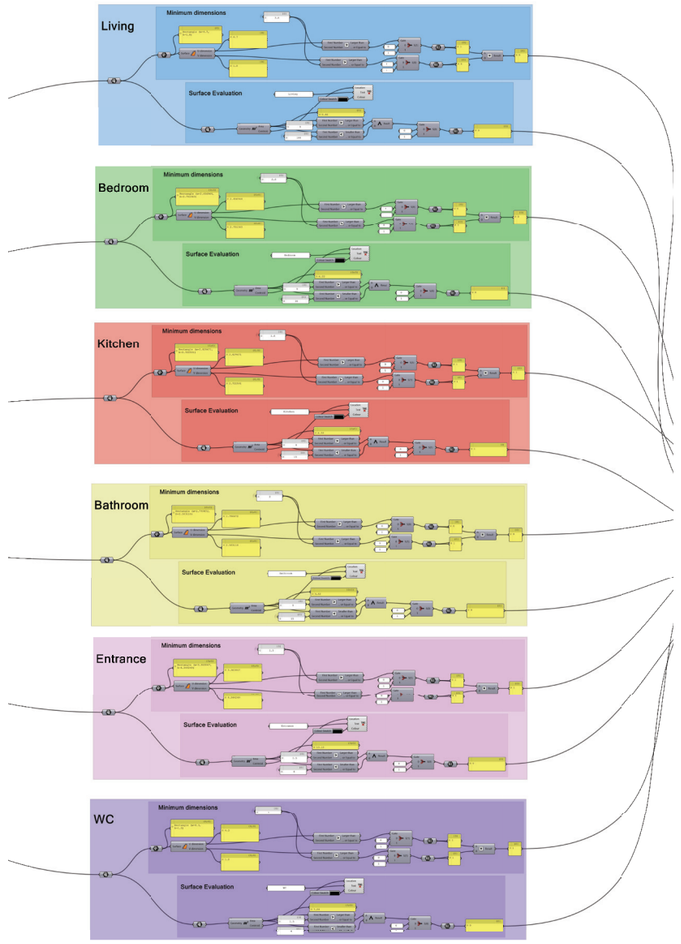


Figure 6.9.a
Partie 9 du programme final
© Arthur ROULAND

Arthur ROULAND

6.10 - Score final de la génération

La dixième partie de cet algorithme permet de réunir l'ensemble des scores obtenus pour le respect de la dimension minimale et l'ensemble des scores pour le respect de la surface. Nous avons donc deux fois six points donc un score maximal de 12 points. L'ensemble des scores obtenus dans chaque catégorie sont additionnés afin de visualiser le score général pour le respect de la dimension minimale et le score général pour le respect de la surface. Ensuite, ces deux scores sont additionnés afin d'obtenir le score final du plan.

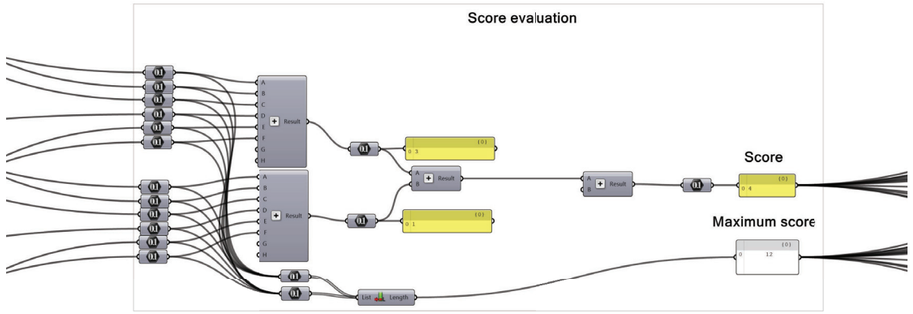


Figure 6.10.a
Partie 10 du programme final
© Arthur ROULAND

6.11 - Dessin du plan

La onzième partie est reliée à la partie 8. Pour rappel, la partie 8 servait à récupérer les contours des surfaces de chaque espace final avant l'évaluation.

Chaque contour va être divisé en plusieurs segments à l'aide du composant "Deconstruct Brep". En utilisant l'option flatten, les courbes ne seront plus rangées dans des listes séparées mais toutes vont appartenir à une et même liste. Le composant "Deconstruct Domain" va nous servir à obtenir le point de départ et le point d'arrivée de chaque segment afin d'en faire des lignes. Dans le premier "Panel" , nous voyons que l'épaisseur des cloisons est renseignée, ici 0.07 mètre soit 7 cm. Cette valeur est divisée par 2 afin d'obtenir la valeur de 3,5 cm. Cette valeur va être multipliée par -1 afin d'obtenir un décalage de notre ligne de 3,5 cm et de -3,5 cm afin que la cloison ait pour axe notre ligne de départ et fasse bien 7 cm. Ces deux lignes sont ensuite transformées en surface grâce au composant "Loft". Si on le souhaite, le composant "Extrusion" permet de fabriquer un modèle 3D du plan généré. La hauteur d'extrusion est paramétrable grâce au slider.

Dans un deuxième temps, les contours de notre plan sont récupérés afin d'y apporter un décalage de 0.2 mètre soit 20 cm. Cela permet de créer une épaisseur plus importante sur le contour du plan.

Enfin, la dernière étape de la partie 11 consiste à assigner une couleur aux surfaces. Afin que les plans soient le plus visible possible, j'ai choisi de leur donner la couleur noire (le fond étant blanc cela créé un fort constraste). Pour cela, j'utilise le composant "Colour CMYK". Dans les entrées "Cyan", "Magenta" et "Yellow" j'entre la valeur 1 afin d'obtenir un noir. Le composant "Create Material" permet de diffuser cette couleur et enfin le composant "Custom Preview" permet d'assigner cette couleur à la géométrie formée par les cloisons et les murs formant le contour du plan.

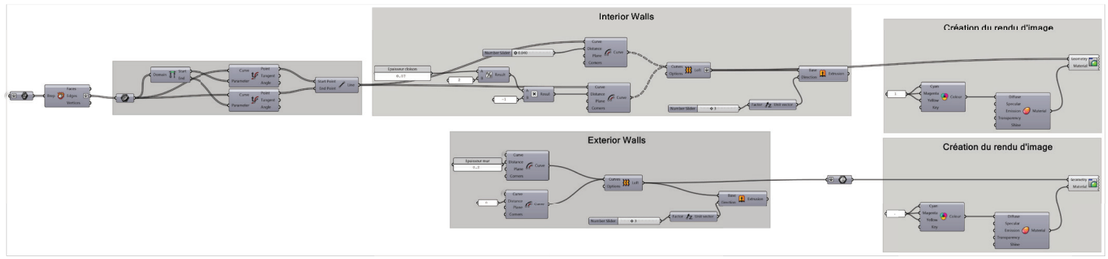


Figure 6.11.a
Partie 11 du programme final
© Arthur ROULAND

Arthur ROULAND

6.12 - Enregistrement de la génération

La douzième et dernière partie de cet algorithme va nous permettre de donner un nom en fonction du score du plan généré et de l'enregistrer dans un dossier sur le disque de l'ordinateur (l'emplacement du fichier est renseigné par l'utilisateur dans la partie 1). Pour les T2, cette partie numéro 12 est divisée en 12 parties pour chacun des 12 points. Pour les T3, elle est composée de 14 parties et pour les T4 de 16 parties.

Tout d'abord, j'utilise le composant "List Length" dans la partie 3 afin d'obtenir le score final maximal souhaité pour chaque plan. 12 points pour un T2, 14 points pour un T3 et 16 points pour un T4. Cette valeur est comparée au score maximal obtenu dans la partie 10. Cette opération est nécessaire car si une ou plusieurs surfaces prioritaires suppriment une surface, alors la pièce n'existe plus. Par exemple, si le séjour et la chambre recouvrent totalement la salle de bain avant la partie 7 (opération booléennes entre les surfaces) alors le score maximal diminue et le plan n'a plus les pièces souhaitées.

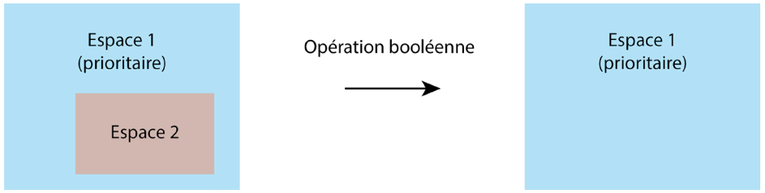


Figure 6.12.a
Schéma de suppression d'une pièce lors d'une opération booléenne
© Arthur ROULAND

J'utilise alors le composant «Equality» afin que le programme sache si le score maximal est bien le bon. Si tel est le cas, la valeur 1 sort de ce composant, sinon la valeur 0 est obtenue.

Le score total de la génération est comparé dans chacune des parties afin de lui attribuer un nom. Ainsi chaque partie a pour entrée une valeur comprise entre 1 et 12 pour les T2. Lorsque le plan généré a un score de 6 points, seul le composant "Equality" se situant dans la partie du score de 6 points va dire que l'égalité est bien respectée et envoyer la valeur 1. Toutes les autres vont envoyer la valeur 0.
Ces deux égalités sont ensuite réunies dans le composant "Gate And". Cette opération permet de vérifier que le score maximal est le bon (qu'il ne manque pas de pièces) et de donner le score du plan comme nom de l'image à enregistrer. Si ces deux paramètres sont respectés alors le composant suivant "Stream Filter" va envoyer la valeur 1. Si l'une de ces deux conditions n'est pas respectée alors il va envoyer la valeur 0. La valeur 1 va activer le script C# permettant l'enregistrement du plan. La valeur 0 ne le permet pas.

Le dernier composant de cet algorithme demande plusieurs entrées. Pour ce programme nous n'utilisons pas l'entrée "trigger".

L'entrée "Dir" nous demande la direction de l'enregistrement. Elles sont déjà toutes préréglées afin d'avoir pour destination ce que l'utilisateur aura précisé dans la partie 1. L'entrée "Name" demande le nom de l'image à enregistrer. Pour chaque score le nom est le suivant "Plan (score) points {0} .png" Chaque plan va donc avoir comme nom son score. Le "{0}" nous permet d'enregistrer les plans avec un nombre comme suffixe afin que chaque nom soit différent et que les plans ne s'écrivent pas les uns sur les autres. Cela aurait pour conséquence de supprimer chaque plan généré à chaque itération. Le ".png" permet de renseigner sur le type de format d'image que l'on souhaite enregistrer.

L'entrée "VP" nous demande le "ViewPort" soit la vue que l'on souhaite enregistrer. Chacun des trois programmes (T2, T3 et T4) a un fichier GrassHopper unique qui doit être ouvert dans le fichier Rhino. Ce fichier Rhino a subi quelques modifications afin de faciliter l'enregistrement des plans. L'environnement Rhino a été rendu blanc afin d'obtenir le contraste le plus fort. Les axes X et Y ainsi que le quadrillage de l'environnement ont été changés en blanc afin de devenir invisibles. Enfin, une vue par défaut a été créée afin d'obtenir des plans ayant les mêmes cadrages et les mêmes proportions.

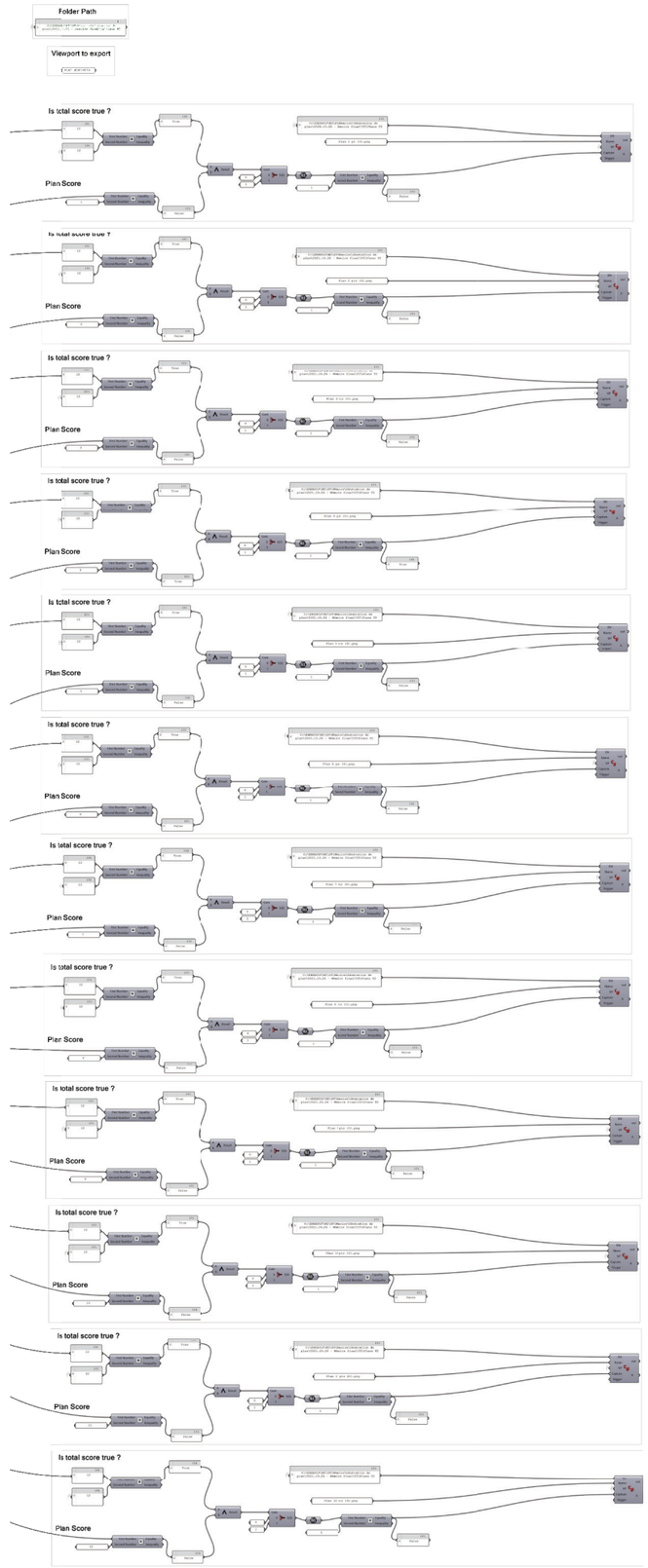
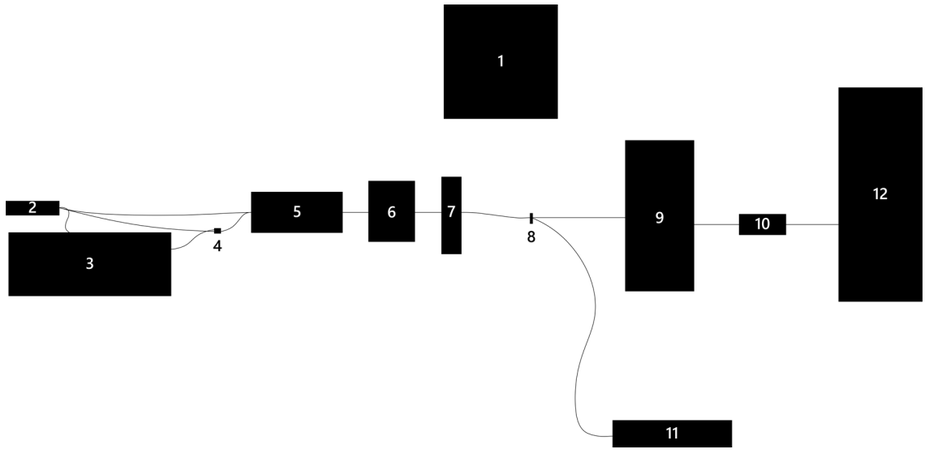


Figure 6.12.b
Partie 12 du programme final
© Arthur ROULAND

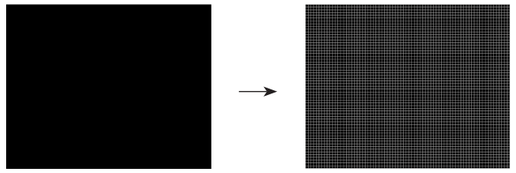
Arthur ROULAND

6.13 - Récapitulatif des étapes

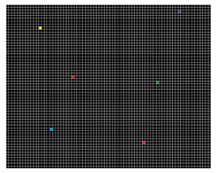


Etape 1 : paramètres de départ

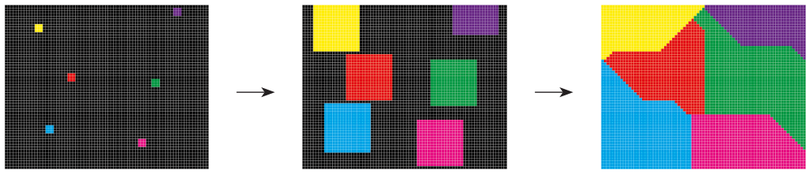
Etape 2 : Définition de la grille



Etape 3 : Valeurs aléatoires de départ

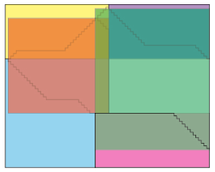


Etape 4 : Division de la grille

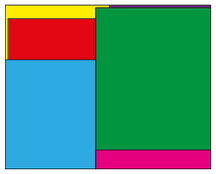


Etape 5 : Attribution des couleurs

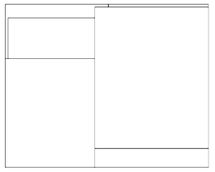
Etape 6 : Simplification des surfaces



Etape 7 : Opérations entre les surfaces



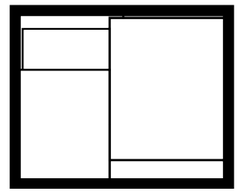
Etape 8 : Récupération des contours



Etape 9 : Evaluation des scores

Etape 10 : Score final de la génération

Etape 11 : Dessin du plan



Etape 12 : Enregistrement de la génération

7

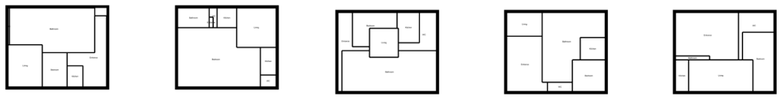
Exemples de plans générés

Arthur ROULAND

7.1 - Génération de T2 (de 1 à 12 points)

1 point

2 points

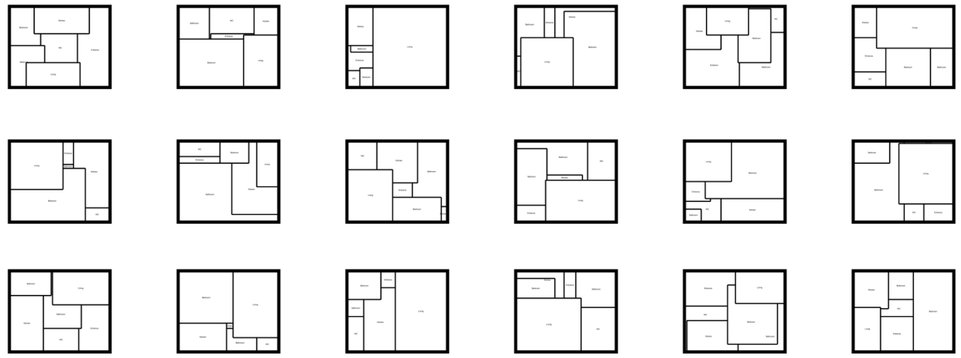


3 points

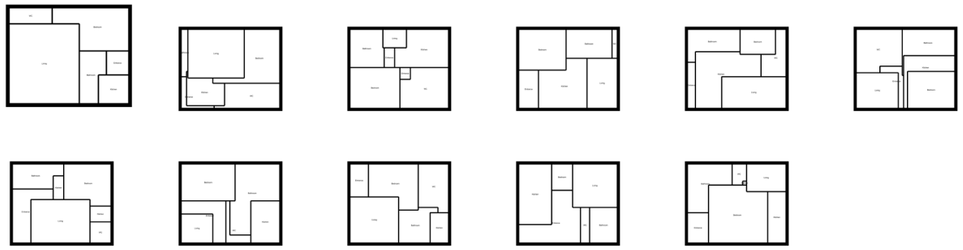
4 points

5 points

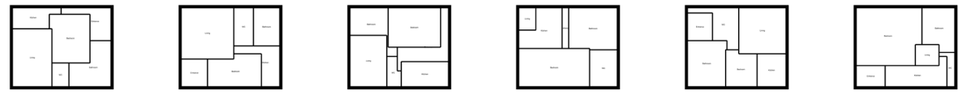
6 points



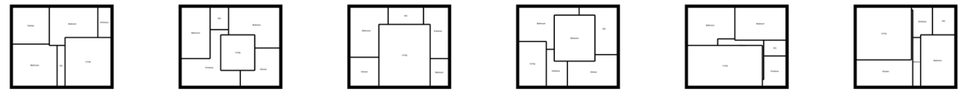
7 points



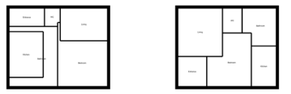
8 points



9 points



10 points



11 points

12 points

Arthur ROULAND

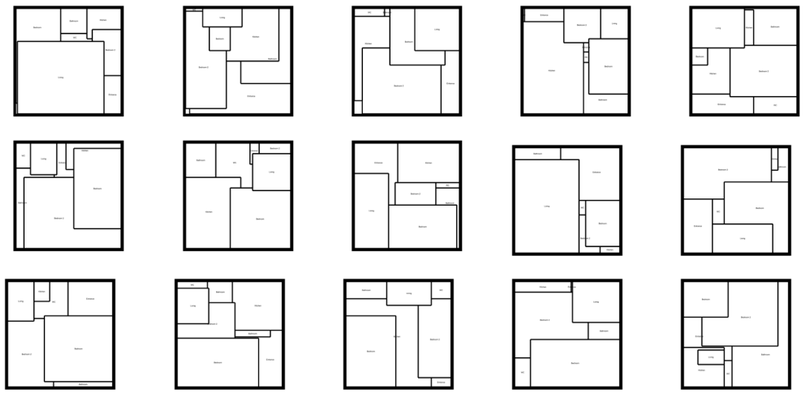
7.2 - Génération de T3 (de 1 à 14 points)

2 points

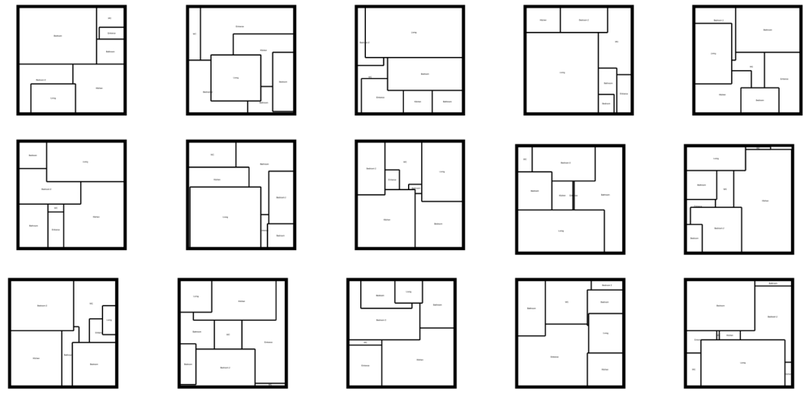
3 points

4 points

5 points



6 points

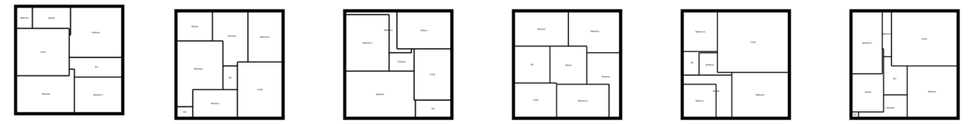


7 points

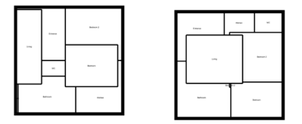
8 points

9 points

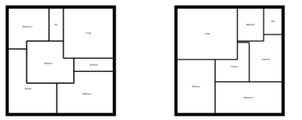
10 points



11 points



12 points



Arthur ROULAND

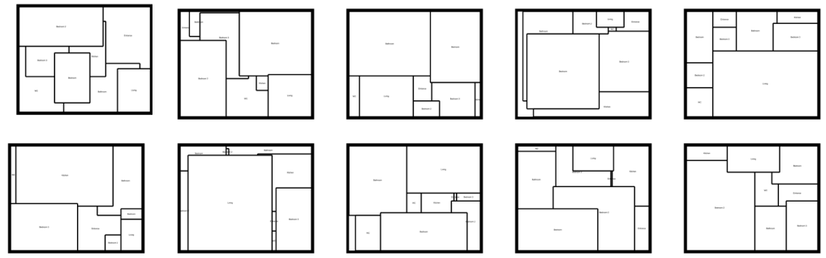
7.3 - Génération de T4 (de 1 à 16 points)

3 points

4 points

5 points

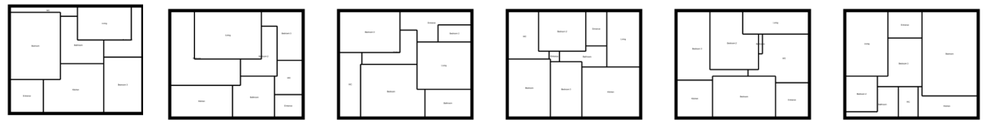
6 points



7 points

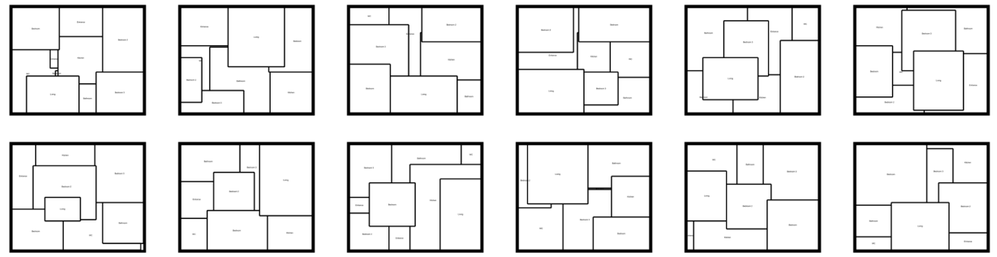
8 points

9 points

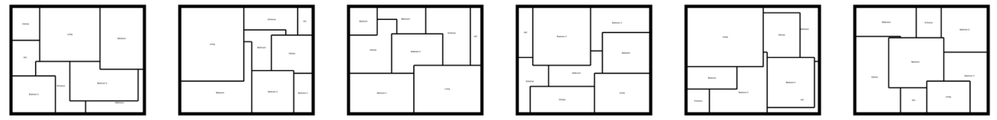


10 points

11 points



12 points



13 points

14 points

15 points

16 points

7.4 - Analyse des résultats

Dans cette partie de ce mémoire, nous allons faire l'analyse des résultats. L'algorithme a tourné pendant de nombreuses heures afin de générer des typologies de plans T2, T3 et T4. Afin de pouvoir comparer les résultats entre les différentes typologies, le nombre de plans générés est sensiblement le même dans chacune des catégories. Pour rappel, les T2 ont un score maximal de 12 points, les T3 ont un score maximal de 14 points et enfin les T4 ont un score maximal de 16 points.

7.4.1 - Résultats de la génération de plans

Typologie de plan	T2	T3	T4
Nombre total de plans générés	300	245	218
Score maximal	12 points	14 points	16 points
1 point	1	0	0
2 points	9	3	0
3 points	20	2	3
4 points	47	23	9
5 points	66	35	19
6 points	70	48	37
7 points	45	51	42
8 points	29	43	44
9 points	11	23	27
10 points	2	13	14
11 points	0	2	16
12 points	0	2	6
13 points		0	1
14 points		0	0
15 points			0
16 points			0

Figure 7.4.1.a
Tableau des résultats de la génération de plans
© Arthur ROULAND

7.4.2 - Pourcentage des scores par typologie

Typologie de plan	T2	T3	T4
1 point	0.33%	0%	0%
2 points	3%	1.22%	0%
3 points	6.67%	0.82%	1.38%
4 points	15.67%	9.38%	4.13%
5 points	22%	14.28%	8.72%
6 points	23.33%	19.59%	16.97%
7 points	15%	20.82%	19.27%
8 points	9.66%	17.55%	20.18%
9 points	3.67%	9.39%	12.38%
10 points	0.67%	5.31%	6.42%
11 points	0%	0.82%	7.34%
12 points	0%	0.82%	2.75%
13 points		0%	0.46%
14 points		0%	0%
15 points			0%
16 points			0%

Figure 7.4.2.a
Tableau des pourcentages des scores par typologie
© Arthur ROULAND

En répertoriant les données dans un tableau nous pouvons déjà apercevoir que la génération semble être plus efficace pour un score moyen. Les graphiques permettent de visualiser cette donnée. En effet, les résultats se voulaient aléatoires afin de générer une diversité infinie de plans. En suivant cette logique, il est normal que le résultat de chacune de ces générations suivent la loi normale.

Définition de la loi normale (par soft-concept.com)

"La courbe de Gauss est connue aussi sous le nom de « courbe en cloche » ou encore de « courbe de la loi normale ». Elle permet de représenter graphiquement la distribution d'une série et en particulier la densité de mesures d'une série. Elle se base sur les calculs de l'espérance et de l'écart-type de la série. Pour un échantillon important, il est généralement constatée une courbe en forme de cloche, c'est-à-dire une forte concentration des valeurs autour de la moyenne puis des valeurs de moins en moins nombreuses aux extrémités de la série."

Arthur ROULAND

La disposition des valeurs de départ de l'algorithme est aléatoire. En ce sens, chacune des couleurs a autant de chance que les autres de tomber sur n'importe quelle case de la grille. L'emplacement initial de ces paramètres de départ influe grandement sur le score de la génération. Il est donc logique que nous n'obtenions que très peu de scores bas ainsi que de scores hauts. Les scores dit "moyens" ont quant à eux beaucoup plus de possibilités d'être générés et c'est pourquoi ils sont majoritaires par rapport aux scores bas et hauts.

Pour simplifier, la loi normale fonctionne comme une paire de dés. Chaque face d'un dé a une chance sur six d'apparaître. Le score le plus bas est donc 2 (1 et 1) et le score le plus haut 12 (6 et 6). Ils ont tous les deux 1 chance sur 36 d'apparaître. Cependant, nous avons plus de chances de faire un score total de 4 par exemple. Les combinaisons possibles sont donc 1 et 3, 2 et 2 et 3 et 1. Cela nous donne 3 chances sur 36 d'obtenir un 3. Ainsi pour les valeurs du milieu comme 7, nous avons alors 6 combinaisons possibles afin d'arriver à ce résultat, soit 6 chances sur 36 ou encore 1 chance sur 6.

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figure 7.4.2.b
Tableau de probabilité pour deux dés à 6 faces
© Arthur ROULAND

La loi de Gauss (ou la loi normale), nous dit que plus la série est grande, donc plus nous allons générer de plans, plus les valeurs vont venir épouser la courbe. Cependant, il est nécessaire de rappeler que l'ensemble de l'algorithme ne repose pas sur le hasard et les lois de probabilités. En effet, le redécoupage des formes générées ainsi que l'ordre de priorité des espaces dans les opérations booléennes peuvent faire varier ce résultat. Cependant, étant donné que ces paramètres restent les mêmes pour chaque plan généré, nous pouvons supposer que ces opérations qui ne relèvent pas de l'aléatoire, n'influent que très peu sur le fait que la série de plans générée suive la loi normale.

7.4.3 - Graphique de la génération des T2

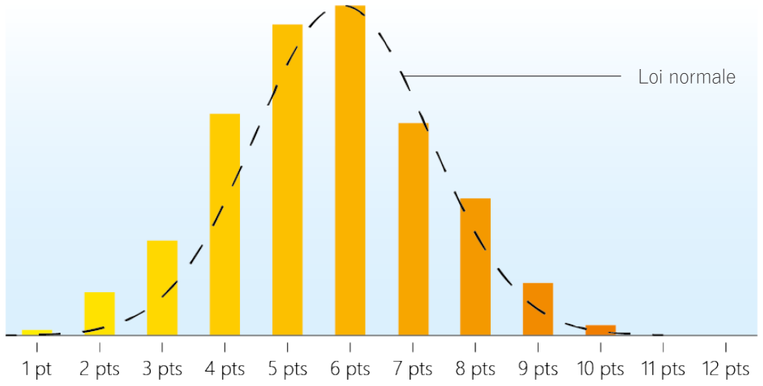


Figure 7.4.3.a
Graphique des résultats de la génération de T2
© Arthur ROULAND

7.4.4 - Graphique de la génération des T3

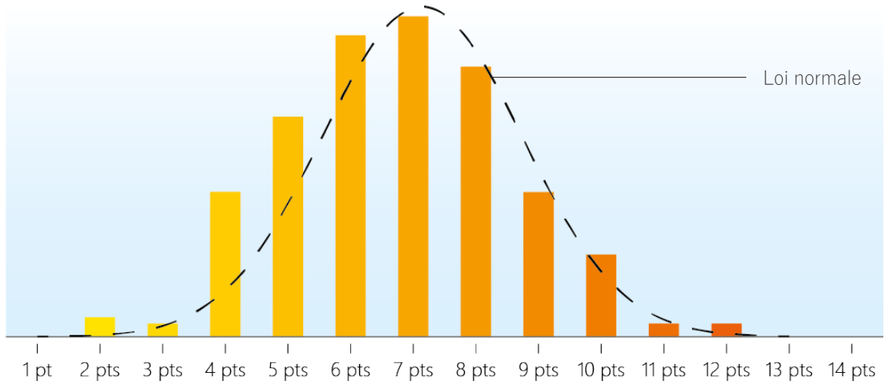


Figure 7.4.4.a
Graphique des résultats de la génération de T3
© Arthur ROULAND

7.4.5 - Graphique de la génération des T4

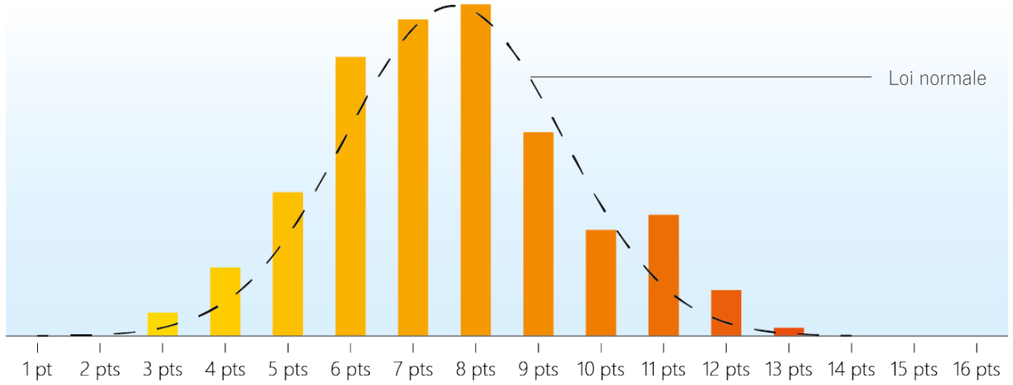


Figure 7.4.5.a
Graphique des résultats de la génération de T4
© Arthur ROULAND

8

Conclusion

Arthur ROULAND

Le résultat de cette recherche est satisfaisant. En effet, l'idée initiale de ce mémoire de recherche était d'élaborer un programme utilisable par tous et permettant de constituer une base de données potentiellement infinie afin de l'utiliser dans le domaine de la recherche en intelligence artificielle. Le programme fonctionne et permet la génération d'une variété infinie de plans comme nous avons pu le voir dans la partie précédente.

Cependant, en effectuant cette recherche, j'ai dû faire des choix qui ont influencé le résultat. En effet, l'ordre des priorités des pièces les unes avec les autres est arbitraire. Les scores et les résultats seraient différents si ce paramètre était modifié. De plus, une des limites de cet algorithme est le lien que les espaces ont avec leur couleur. En effet, dans ce mémoire, chaque couleur était dès le début associée à un espace. Une amélioration à apporter serait de générer une partition impartiale qui évaluerait par la suite les espaces afin d'identifier chaque pièce. Dans cette expérience, il est impossible d'intervertir deux espaces. En ce sens, un séjour qui répond aux critères d'une cuisine et une cuisine qui répond aux critères d'un séjour ne peuvent pas échanger leurs emplacements. Cela constitue une limite à cette expérience car un plan spatialement intéressant avec cette affectation de pièces peut se retrouver avec un score faible tandis que si l'on avait interverti deux espaces entre eux, le score aurait pu être plus élevé.

Un autre paramètre important que nous avons déjà abordé dans ce mémoire est la division de la grille. En effet, beaucoup de tentatives ont été nécessaires afin de trouver une solution qui offrait un vaste domaine de possibilités. Le choix des automates cellulaires a été fait dans le cadre de cette recherche mais d'autres méthodes peuvent également être développées qui répondent ou non aux enjeux que je me suis fixé pour ce mémoire. Une autre méthode de division d'une surface engendrera sans aucun doute des résultats différents et constituera une base de données différentes pour la recherche dans le domaine de l'intelligence artificielle.

Une des limites à laquelle j'ai dû faire face est la forme du plan généré. Dans la partie 5, nous avons pu voir plusieurs méthodes de division d'une surface. J'ai pu essayer le composant Substrate qui est incapable de fonctionner sur une autre forme qu'un rectangle. La méthode basée sur l'utilisation des automates cellulaires présente actuellement dans ce mémoire la même limite. En effet, même en dessinant une surface plus complexe qu'un rectangle, l'automate redessine une forme rectangulaire. Il doit cependant être possible d'adapter ce programme à une forme plus complexe afin de permettre la génération d'une plus grande diversité de plans.

Enfin, un des derniers choix que j'ai pu faire influence également le générateur. En effet, un des derniers composants ajoutés dans le programme GrassHopper est un système d'évaluation du score final. Cette partie de l'algorithme compare le score final maximal possible et le score maximal final de la génération. En effet, la partie permettant d'effectuer des opérations booléennes entre les surfaces peut parfois effacer un espace. Dans ce cas précis, la pièce disparaît et une partie de l'algorithme dysfonctionne. Le résultat de ce phénomène est que le score final de la génération n'est pas en adéquation avec le score maximal possible. J'ai pris la décision arbitraire de ne pas enregistrer les plans qui auraient un espace manquant dans le but de favoriser la qualité à la quantité. J'ai pu remarquer que plus il y avait de pièces différentes dans un plan, plus ce phénomène pouvait être récurrent. Les trois algorithmes pour les T2, T3 et T4 sont tous les trois réglés sur 20 secondes. Ils génèrent donc théoriquement un plan toutes les 20 secondes (180 plans à l'heure) mais tous ne sont pas enregistrés afin d'éviter d'obtenir trop d'anomalies et de se retrouver parfois avec des plans entièrement blancs.

Pour terminer, contrairement aux plug-ins Marmot et Magnetizing Floor Plan Generator, les pièces n'entretiennent pas de relations entre elles. Aucune logique de partition n'est établie et nous pouvons parfois observer des pièces aveugles enclavées dans le plan. Cela n'est pas forcément mauvais car il est intéressant d'observer ces anomalies. Les plans générés n'ont pas d'orientation particulière, d'ouvertures particulières et l'on peut aussi bien s'imaginer dans un immeuble des années 50 que dans une tiny-house dans la nature. L'idée était de générer et d'observer une grande diversité de plans afin d'obtenir parfois des anomalies, des choses qui sortent de l'ordinaire qui ne sont pas conventionnelles mais qui peuvent fonctionner et donner des idées nouvelles. Jean-Raphaël PIQUARD a lui aussi généré des anomalies ou plutôt des «chimères». Je pensais en obtenir après avoir testé son programme avec ma base de données mais mon programme de génération de plan génère lui aussi des «chimères» ce qui est intéressant.

9

Notice d’utilisation

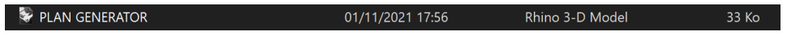
Arthur ROULAND

9 - Notice d'utilisation

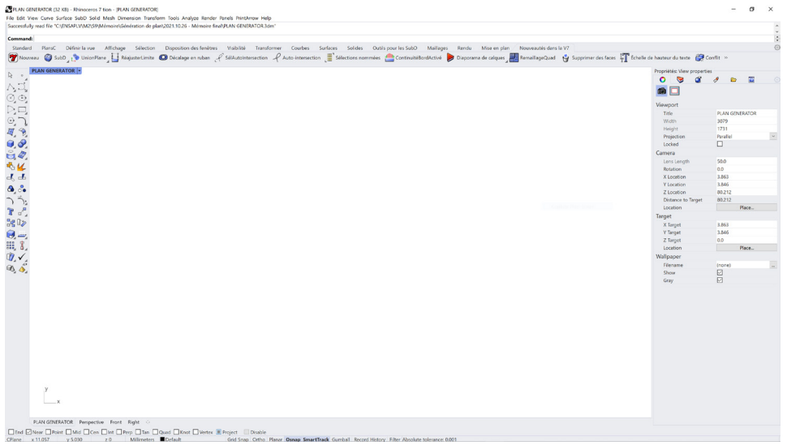
1 - Installez Rhinoceros 3D version 6 ou 7 avec Grasshopper.

2 - Téléchargez le fichier et installez-le sur le disque dur de votre ordinateur.

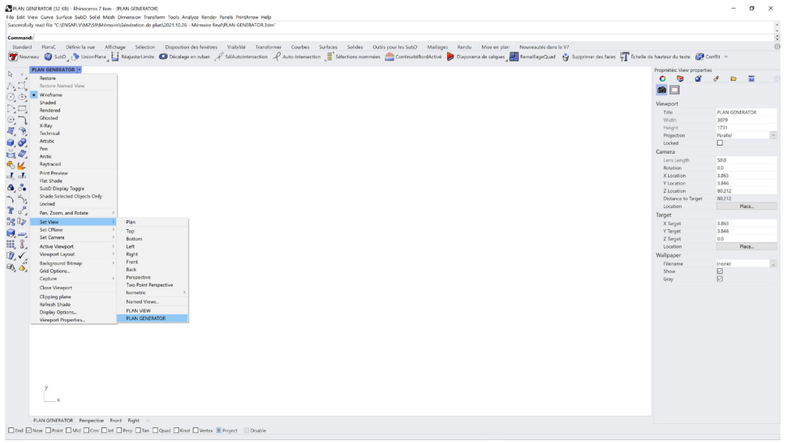
3 - Ouvrez le fichier Rhino "PLAN GENERATOR.3dm".



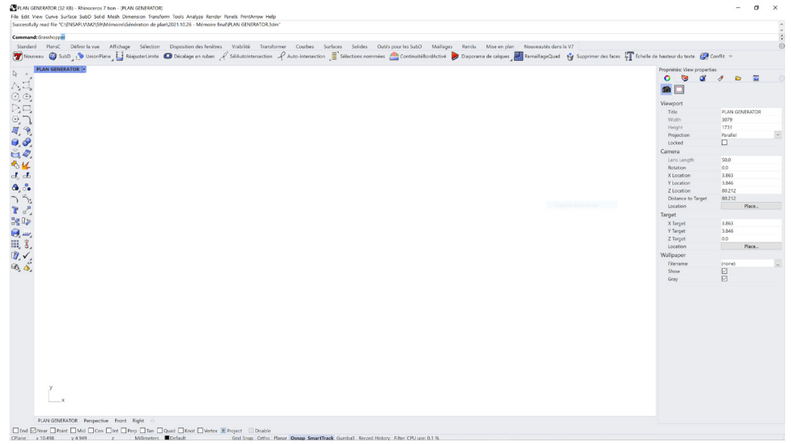
4 - Vérifiez que le fond s'affiche en blanc.



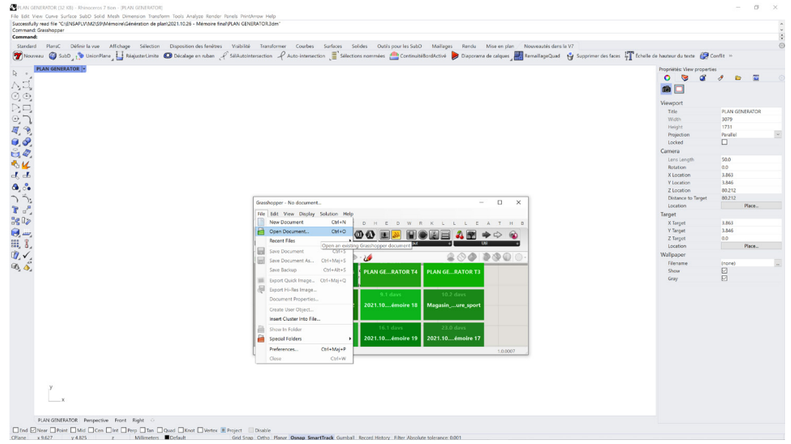
5 - Vérifiez que la vue courante est la vue "PLAN GENERATOR". Cliquer sur la flèche à droite du nom de la vue. Se rendre dans "Set View" puis cliquez sur "PLAN GENERATOR".



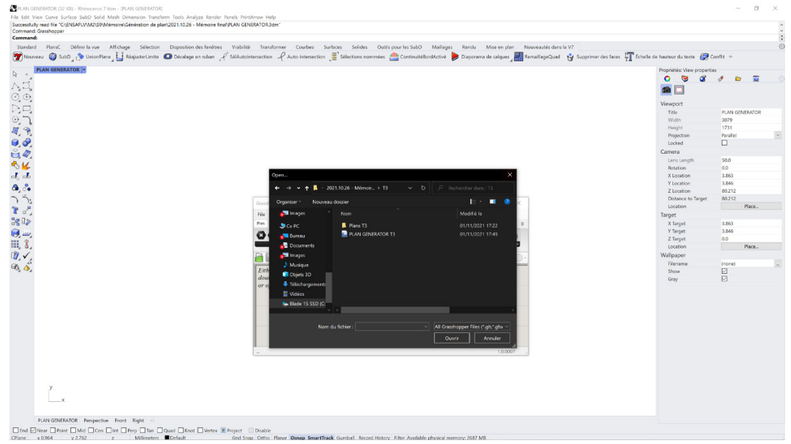
6 - Dans la partie "Command" de Rhino, entrer le mot "Grasshopper", puis cliquer sur "Enter".



7 - Cliquez sur "File" puis "Open Document".

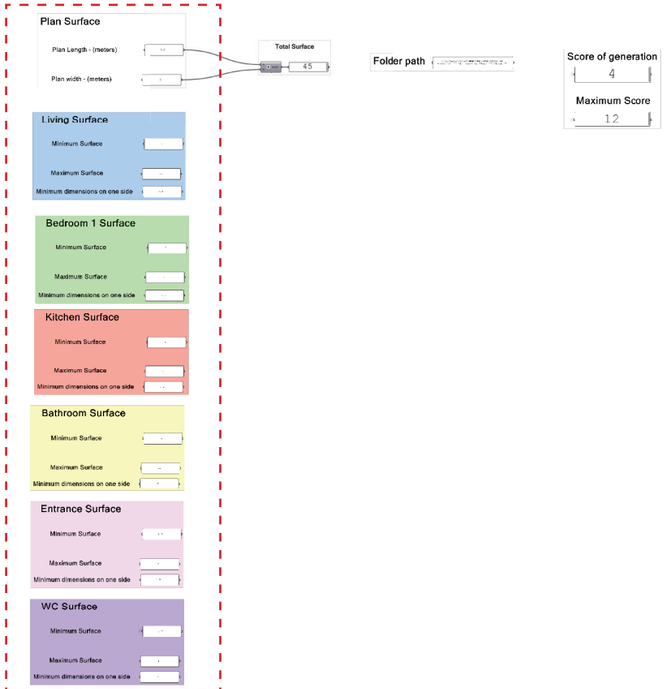


8 - Allez dans le fichier téléchargé, entrez dans le dossier que vous souhaitez générer (T2, T3 ou T4) et lancez le fichier Grasshopper "PLAN GENERATOR T···".

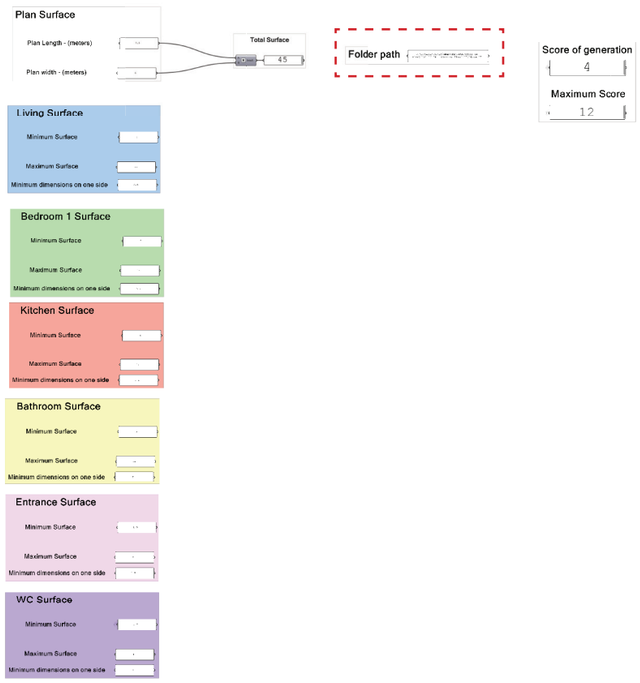


Arthur ROULAND

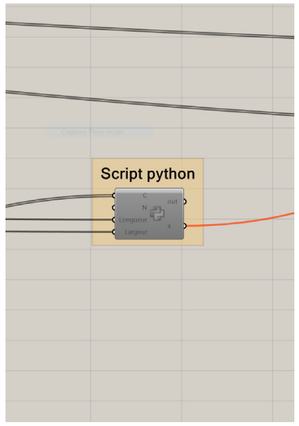
9 - Dans la partie supérieure du programme (partie 1), rentrez les paramètres du plan. Longueur et largeur du plan, surface minimale, maximale et dimensions minimales des côtés de chaque pièce. Le tout en mètres. Utilisez des points plutôt que des virgules.

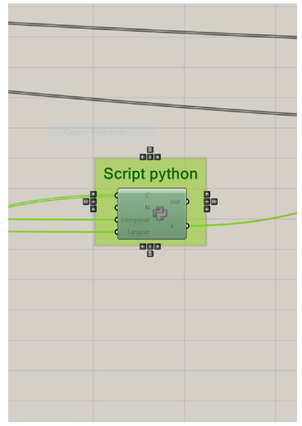


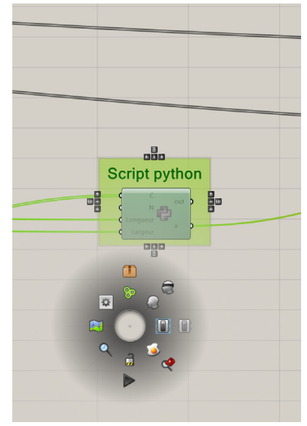
10 - Toujours dans la partie 1, entrez le chemin d'accès au fichier où vous souhaitez enregistrer les plans générés à chaque itération.



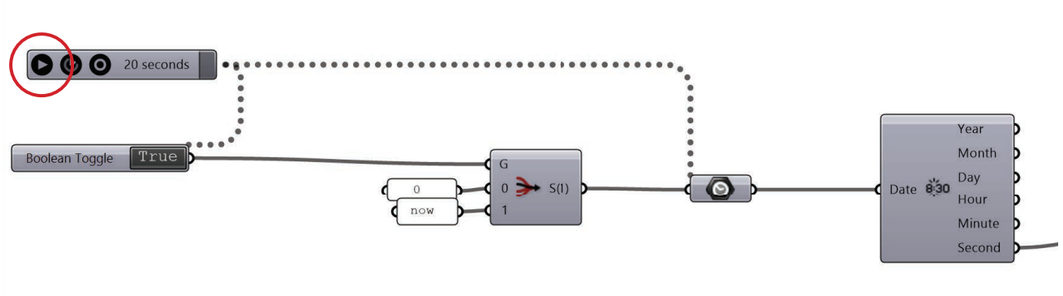
11 - Par défaut, le script Python est éteint. Sélectionnez le. Appuyez sur la molette de votre souris. Cliquez sur le bouton "On". Un premier plan devrait être généré.







12 - Allez au début de la partie 2 (tout à gauche du programme). Lancez le timer. Par défaut, le timer génère de nouvelles valeurs toutes les 20 secondes. Cet intervalle est à régler en fonction des performances de votre ordinateur.



10

Bibliographie

Arthur ROULAND

TRAVAUX DE RECHERCHE

• **BEN ABDALLAH Yasmine**

"Conception architecturale et modélisation paramétrique", Mémoire de recherche, Ecole Nationale Supérieure d'Architecture de Toulouse, 2017

Disponible à l'adresse : https://dumas.ccsd.cnrs.fr/dumas-01807920/document

Consulté en Octobre 2021

• **BEYLER Nathan**

"Utiliser et transformer des critères en esquisses de plan ", Mémoire de recherche, Ecole Nationale Supérieure d'Architecture de Paris La Villette, 2019

Consulté le 4 Mai 2021

• **CARADANT Dominique**

"Les utilitaires et l'intelligence artificielle pour un système d'aide à la conception en architecture", Rapport de recherche, Ecole Nationale Supérieure d'Architecture de Toulouse, Juin 1984.

Disponible à l'adresse : https://hal.archives-ouvertes.fr/hal-01888604/document

Consulté le 23 Novembre 2020

• **CARTA Silvio**

"Self-organizing Floor Plans", Article de recherche, Head of Design, University of Hertfordshire, UK, Publié dans le Harvard Data Science Review, 23 Juillet 2021

Disponible à l'adresse : https://assets.pubpub.org/cjin385t/51627069693624.pdf

• **CARTA Silvio & ST LOE Stephanie**

"Self-Organising Floor Plans in Care Homes", Papier de recherche, University of Hertfordshire, UK, Publication en ligne 27 Mai 2020, Publié dans le journal Sustainability le 1er Juin 2020

Disponible à l'adresse : https://www.researchgate.net/publication/341720764_Self-Organising_Floor_Plans_in_Care_Homes

• **CHAILLOU Stanislas**

AI + Architecture, Towards a New Approach", Thèse, Harvard Graduate School of Design, 2019.

Disponible à l'adresse : http://stanislaschaillou.com/articles.html

Consulté en Mars 2020

• **HERR Christiane M. & FORD Ryan C.**

"ADAPTING CELLULAR AUTOMATA AS ARCHITECTURAL DESIGN TOOLS", Papier de recherche, Xi'an Jiaotong-Liverpool University, Suzhou, China, 2015, Publié dans le Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA) en 2015.

Disponible à l'adresse : http://papers.cumincad.org/data/works/att/caadria2015_139.content.pdf

• **KRAWCZYK Robert J.**

"Architectural Interpretation of Cellular Automata", Papier de recherche, Illinois Institute of Technology, Pour l'International Conference on Generative Art, 2002 et publié sur https://www.generativeart.com/ en 2002

Disponible à l'adresse : https://www.researchgate.net/publication/2856485_Architectural_Interpretation_of_Cellular_Automata

• **PIQUARD Jean-Raphaël**

"L'apprentissage machine au service de la conception architecturale", Mémoire de recherche, Ecole Nationale Supérieure d'Architecture de Paris La Villette, 2020

Consulté en Décembre 2020

• **SALINI Alex**

"L'utilisation d'un réseau adversarial antagoniste (GAN) dans la création de plans d'architecture", Mémoire DPEA, Ecole Nationale Supérieure d'Architecture de Paris La Villette laboratoire MAP MAACC, 2020

Consulté en Décembre 2020

• **SHEKHAWAT Krishnendra**

"Automated space allocation using mathematical techniques", 1er Avril 2015, Department of Mathematics, University of Geneva, Switzerland, Publié dans le Ain Shams Engineering Journal en 2015.

Disponible à l'adresse : https://www.sciencedirect.com/science/article/pii/S2090447915000349#:~:text=Space%20allocation%20is%20the%20computational,and%20topological%20and%20geometric%20constraints

• **SHEKHAWAT Krishnendra**

"Space Allocation in Rectangular Floor Plan", Thèse, Décembre 2012, Department of Mathematics, University of Geneva, Switzerland, Publié dans ELSA International Review en 2013.

Disponible à l'adresse : https://www.researchgate.net/publication/262152582_Space_Allocation_in_Rectangular_floor_Plan

• **Zifeng Guo**

"Evolutionary approach for spatial architecture layout design enhanced by an agent-based topology finding system", Papier de recherche, Janvier 2017, School of Architecture, Southeast University, Nanjing 210096, Chine, Publié dans le journal Frontiers of Architectural Research en 2017.

Disponible à l'adresse : https://www.researchgate.net/publication/312263676_Evolutionary_approach_for_spatial_architecture_layout_design_enhanced_by_an_agent-based_topology_finding_system

• **Zifeng Guo**

"Generated Building Layout", date de publication inconnue.

Disponible à l'adresse : https://www.academia.edu/14081792/Generated_Building_Layout

Consulté en Décembre 2021

RESSOURCES EN LIGNE

• **BIELIK Martin**

"Magnetizing Floor Plan Generator", Article, 23 Mars 2019

Disponible à l'adresse : https://toolbox.decodingspaces.net/magnetizing-floor-plan-generator/

Consulté en Septembre 2021

• **GUENA François**

"TR707 Initiation à la modélisation paramétrique", Cours de Technique de Représentation, ENSAPLV

Disponible à l'adresse : http://maacc.paris-lavillette.archi.fr/TR707/

Consulté en Janvier 2021

• **GUENA François**

Cours 4 &5 de **"CTID 825 : Systèmes numériques de la conception digitale"**, ENSAPLV

Disponible à l'adresse : http://194.199.196.229/CTID823/

Consulté en Juin 2021

VIDÉOS

• **CHAILLOU Stanislas**

"IA & Architecture", Conférence au Pavillon de l'arsenal, 27 Février 2020.

Disponible à l'adresse : https://www.facebook.com/PavillonArsenal/videos/889535584839133/

Consulté en Mars 2020

• **GAVRILOV Egor**

"Magnetizing FloorPlanGenerator preview 1", Vidéo YouTube, 9 Mars 2019

Disponible à l'adresse : https://www.youtube.com/watch?v=VWQg8BtrbNU&ab_channel=EgorGavrilov

Consulté en Septembre 2021

• **GAVRILOV Egor**

"Magnetizing FloorPlanGenerator preview 2", Vidéo YouTube, 9 Mars 2019

Disponible à l'adresse : https://www.youtube.com/watch?v=glajwXOuU0I&ab_channel=EgorGavrilov

Consulté en Septembre 2021

• **Yannick M**

"Marmot demo video", Vidéo YouTube, 4 Janvier 2020

Disponible à l'adresse : https://www.youtube.com/watch?v=XTwI6iR9_J4&ab_channel=YannickM

Consulté en Septembre 2021

FICHIERS

• **FOTADIS Peter**

"Randoms_Unique_viaHashSet_V1.gh", Rhinoceros Forums, Real random numbers, 25 Février 2018.

Disponible à l'adresse : https://discourse.mcneel.com/t/real-random-numbers/56072/5

Consulté le 13 Octobre 2021

• **RUTTEN David**

"capture.gh", Rhinoceros Forums, Capturing Rhino layout/viewport iterations - print or image export, 12 Décembre 2017.

Disponible à l'adresse : https://discourse.mcneel.com/t/capturing-rhino-layout-viewport-iterations-print-or-image-export/51387

Consulté le 12 Octobre 2021

• **Wallgren Arkitekter and BOX Bygg**

"Finch 3D", 2019

Disponible à l'adresse : https://finch3d.com/

Disponible à l'adresse : https://www.archdaily.com/929300/can-a-machine-perform-the-work-of-an-architect-a-chat-with-jesper-wallgren-founder-at-finch-3d

Consulté en 2020

11

Glossaire

Arthur ROULAND

Add-on : Logiciel conçu pour être greffé à un autre logiciel à travers une interface prévue à cet effet, et apporter à ce dernier de nouvelles fonctionnalités.

Algorithme : Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations. Un algorithme peut être traduit, grâce à un langage de programmation, en un programme exécutable par un ordinateur.

Allocation spatiale : Disposition informatique des pièces dans un plan. C'est le processus de détermination de la position et de la taille de différentes pièces dans un espace à deux dimensions, en fonction des exigences de l'utilisateur et des contraintes topologiques et géométriques.

Automate cellulaire : Un automate cellulaire consiste en une grille régulière de « cellules » contenant chacune un « état » choisi parmi un ensemble fini et qui peut évoluer au cours du temps.

CAO/CAD : Conception Assistée par Ordinateur / Computer-Aided Design

Day & Night : Automate cellulaire bidimensionnel à deux états (« vivant » ou « mort »)

DWG : Format natif des fichiers de dessins AutoCAD. DWG est l'abréviation de DraWinG

GrassHopper : Grasshopper est un langage et un environnement de programmation visuel qui s'exécute dans l'application de conception assistée par ordinateur (CAO) Rhinoceros 3D

Intelligence Artificielle : Ensemble de théories et de techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine.

Itération : Répétition d'un calcul, d'une opération, d'un raisonnement.

Jeu de la vie : Automate cellulaire imaginé par John Horton Conway en 1970. Malgré des règles très simples, le jeu de la vie permet le développement de motifs extrêmement complexes.

Logiciel : Ensemble de programmes, qui permet à un ordinateur ou à un système informatique d'assurer une tâche ou une fonction en particulier.

Opération booléenne : Ensemble d'opérations booléennes (AND, OR, NOT, XOR...) effectuées sur un ou plusieurs ensembles de polygones en infographie.

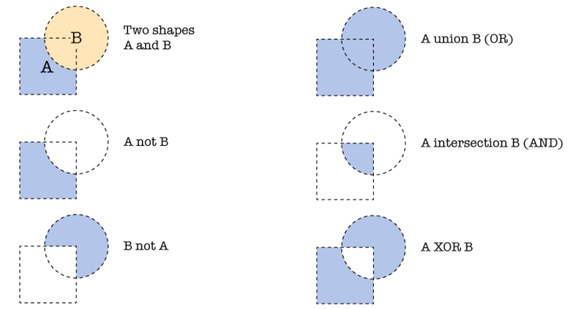


Figure 11.a
Schémas d'opérations booléennes
Source : wikipédia.org

Paramétrique : Qui contient un ou des paramètres, c'est à dire une variable susceptible de recevoir une valeur constante pour un cas déterminé et qui désigne certains coefficients ou certaines quantités en fonction desquels on veut exprimer une proposition ou les solutions d'un système d'équations.

Plug-in : Logiciel conçu pour être greffé à un autre logiciel à travers une interface prévue à cet effet, et apporter à ce dernier de nouvelles fonctionnalités.

PNG : Portable Network Graphics. Format d'image numérique.

Python : Langage de programmation multiplateforme.

Rhinocéros 3D : Logiciel 3D de Conception Assistée par Ordinateur permettant la modélisation d'objets complexes.

Système multi-agents : Système composé d'un ensemble d'agents (un processus, un robot, un être humain, une fourmi etc.), actifs dans un certain environnement et interagissant selon certaines règles.

Voronoï : En mathématiques, un diagramme de Voronoï est un pavage (découpage) du plan en cellules (régions adjacentes) à partir d'un ensemble discret de points appelés « germes ». Chaque cellule renferme un seul germe, et forme l'ensemble des points du plan plus proches de ce germe que d'aucun autre. La cellule représente en quelque sorte la « zone d'influence » du germe.

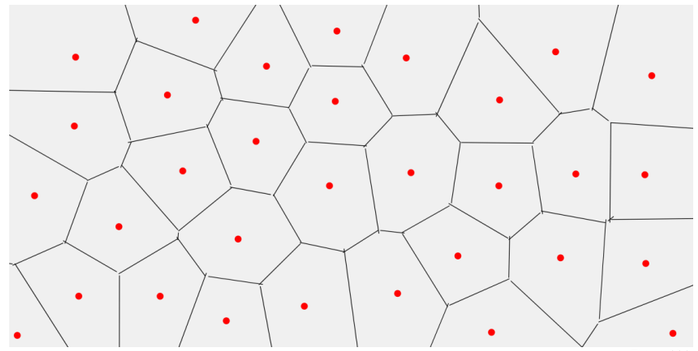


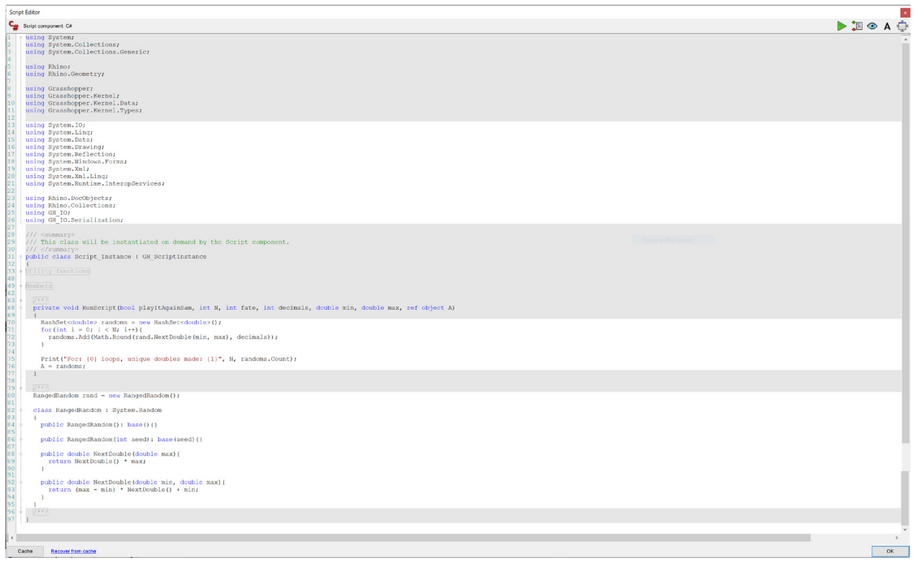
Figure 11.b
Voronoï
Source : javalab.org

12

Annexes

Arthur ROULAND

12.1 - Capture d'écran du timer



12.1.2 - Code du timer

```
using System;
using System.Collections;
using System.Collections.Generic;

using Rhino;
using Rhino.Geometry;

using Grasshopper;
using Grasshopper.Kernel;
using Grasshopper.Kernel.Data;
using Grasshopper.Kernel.Types;

using System.IO;
using System.Linq;
using System.Data;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using System.Xml;
using System.Xml.Linq;
using System.Runtime.InteropServices;

using Rhino.DocObjects;
using Rhino.Collections;
using GH_IO;
using GH_IO.Serialization;

/// <summary>
/// This class will be instantiated on demand by the Script component.
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
#region Utility functions
  /// <summary>Print a String to the [Out] Parameter of the Script component.</summary>
  /// <param name="text">String to print.</param>
  private void Print(string text) { /* Implementation hidden. */ }
  /// <summary>Print a formatted String to the [Out] Parameter of the Script component.</summary>
  /// <param name="format">String format.</param>
  /// <param name="args">Formatting parameters.</param>
  private void Print(string format, params object[] args) { /* Implementation hidden. */ }
  /// <summary>Print useful information about an object instance to the [Out] Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj) { /* Implementation hidden. */ }
  /// <summary>Print the signatures of all the overloads of a specific method to the [Out] Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj, string method_name) { /* Implementation hidden. */ }
#endregion

#region Members
  /// <summary>Gets the current Rhino document.</summary>
  private readonly RhinoDoc RhinoDocument;
  /// <summary>Gets the Grasshopper document that owns this script.</summary>
  private readonly GH_Document GrasshopperDocument;
  /// <summary>Gets the Grasshopper script component that owns this script.</summary>
  private readonly IGH_Component Component;
  /// <summary>
  /// Gets the current iteration count. The first call to RunScript() is associated with Iteration==0.
  /// Any subsequent call within the same solution will increment the Iteration count.
  /// </summary>
  private readonly int Iteration;
#endregion

  /// <summary>
  /// This procedure contains the user code. Input parameters are provided as regular arguments,
  /// Output parameters as ref arguments. You don't have to assign output parameters,
  /// they will have a default value.
  /// </summary>
  private void RunScript(bool playItAgainSam, int N, int fate, int decimals, double min, double max, ref object A)
  {
    HashSet<double> randoms = new HashSet<double>();
    for(int i = 0; i < N; i++){
      randoms.Add(Math.Round(rand.NextDouble(min, max), decimals));
    }

    Print("For: {0} loops, unique doubles made: {1}", N, randoms.Count);
    A = randoms;
  }

  // <Custom additional code>
  RangedRandom rand = new RangedRandom();

  class RangedRandom : System.Random
  {
    public RangedRandom(): base(){}

    public RangedRandom(int seed): base(seed){}

    public double NextDouble(double max){
      return NextDouble() * max;
    }

    public double NextDouble(double min, double max){
      return (max - min) * NextDouble() + min;
    }
  }
  // </Custom additional code>
}
```

Arthur ROULAND

12.2 - Capture d'écran du script Python

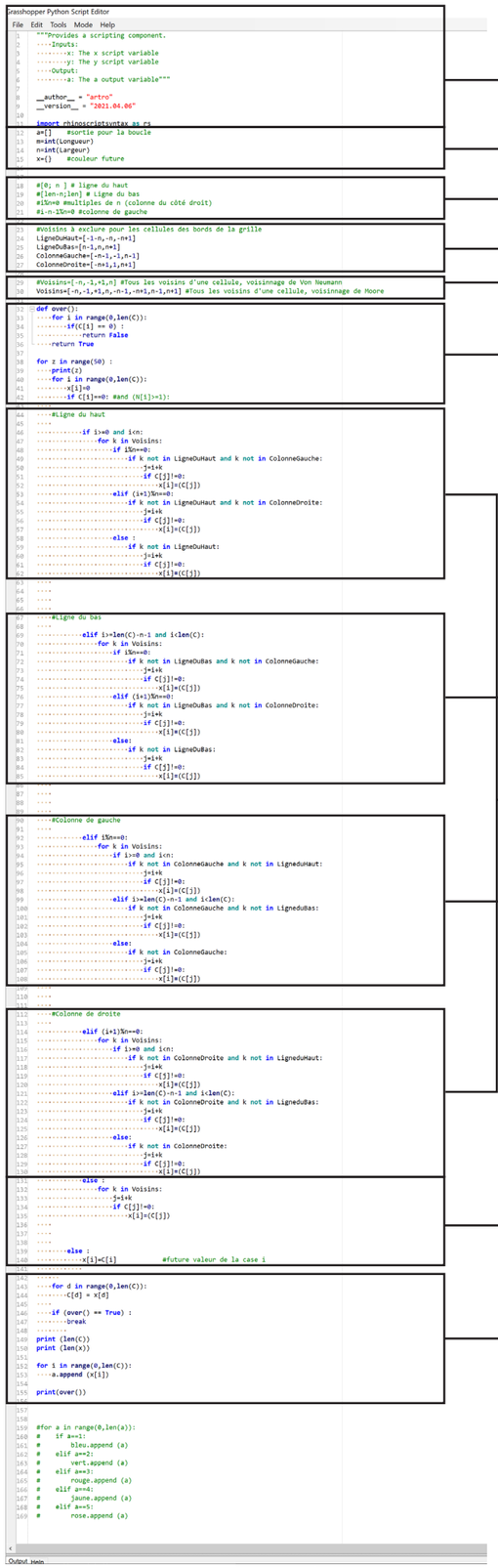

```
"""Provides a scripting component.
    Inputs:
        x: The x script variable
        y: The y script variable
    Output:
        a: The a output variable"""

__author__ = "artro"
__version__ = "2021.04.06"

import rhinoscriptsyntax as rs
a=[]    #sortie pour la boucle
m=int(Longueur)
n=int(Largeur)
x={}    #couleur future


#[0; n ] # ligne du haut
#[len-n;len] # Ligne du bas
#i%n=0 #multiples de n (colonne du côté droit)
#i-n-1%n=0 #colonne de gauche

#Voisins à exclure pour les cellules des bords de la grille
LigneDuHaut=[-1-n,-n,-n+1]
LigneDuBas=[n-1,n,n+1]
ColonneGauche=[-n-1,-1,n-1]
ColonneDroite=[-n+1,1,n+1]

#Voisins=[-n,-1,+1,n] #Tous les voisins d'une cellule, voisinnage de Von Neumann
Voisins=[-n,-1,+1,n,-n-1,-n+1,n-1,n+1] #Tous les voisins d'une cellule, voisinnage de Moore

def over():
    for i in range(0,len(C)):
        if(C[i] == 0) :
            return False
    return True

for z in range(50) :
    print(z)
    for i in range(0,len(C)):
        x[i]=0
        if C[i]==0: #and (N[i]>=1):

    #Ligne du haut

            if i>=0 and i<n:
                for k in Voisins:
                    if i%n==0:
                        if k not in LigneDuHaut and k not in ColonneGauche:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    elif (i+1)%n==0:
                        if k not in LigneDuHaut and k not in ColonneDroite:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    else :
                        if k not in LigneDuHaut:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])



    #Ligne du bas

            elif i>=len(C)-n-1 and i<len(C):
                for k in Voisins:
                    if i%n==0:
                        if k not in LigneDuBas and k not in ColonneGauche:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    elif (i+1)%n==0:
                        if k not in LigneDuBas and k not in ColonneDroite:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    else:
                        if k not in LigneDuBas:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
```


12.2.2 - Code du script Python

Arthur ROULAND

```
#Colonne de gauche

    elif i%n==0:
        for k in Voisins:
            if i>=0 and i<n:
                if k not in ColonneGauche and k not in LigneduHaut:
                    j=i+k
                    if C[j]!=0:
                        x[i]=(C[j])
            elif i>=len(C)-n-1 and i<len(C):
                if k not in ColonneGauche and k not in LigneduBas:
                    j=i+k
                    if C[j]!=0:
                        x[i]=(C[j])
            else:
                if k not in ColonneGauche:
                    j=i+k
                    if C[j]!=0:
                        x[i]=(C[j])


#Colonne de droite

    elif (i+1)%n==0:
        for k in Voisins:
            if i>=0 and i<n:
                if k not in ColonneDroite and k not in LigneduHaut:
                    j=i+k
                    if C[j]!=0:
                        x[i]=(C[j])
            elif i>=len(C)-n-1 and i<len(C):
                if k not in ColonneDroite and k not in LigneduBas:
                    j=i+k
                    if C[j]!=0:
                        x[i]=(C[j])
            else:
                if k not in ColonneDroite:
                    j=i+k
                    if C[j]!=0:
                        x[i]=(C[j])
    else :
        for k in Voisins:
            j=i+k
            if C[j]!=0:
                x[i]=(C[j])



    else :
        x[i]=C[i]           #future valeur de la case i


for d in range(0,len(C)):
    C[d] = x[d]

if (over() == True) :
    break

print (len(C))
print (len(x))

for i in range(0,len(C)):
    a.append (x[i])

print(over())


#for a in range(0,len(a)):
#    if a==1:
#        bleu.append (a)
#    elif a==2:
#        vert.append (a)
#    elif a==3:
#        rouge.append (a)
#    elif a==4:
#        jaune.append (a)
#    elif a==5:
#        rose.append (a)
```


12.2.3 - Explication du script python



12.2.3.1 - Le script codé par grasshopper

Grasshopper Python Script Editor

File Edit Tools Mode Help

```
"""Provides a scripting component.
    Inputs:
        x: The x script variable
        y: The y script variable
    Output:
        a: The a output variable"""

__author__ = "artro"
__version__ = "2021.04.06"

import rhinoscriptsyntax as rs
```

La première partie de ce script python est déjà codée par défaut lors de l'ouverture du composant. Aucune modification n'a été apportée.

12.2.3.2 - Définitions des variables

```
import rhinoscriptsyntax as rs
a=[]    #sortie pour la boucle
m=int(Longueur)
n=int(Largeur)
x={}    #couleur future

```

Dans cette deuxième partie du script, nous paramétrons les deux entrées "Longueur" et "Largeur" du composant ainsi que sa sortie "a". C'est dans cette sortie que vont être stockées les informations finales. A celà nous avons rajouté une variable "x". Cette variable va nous permettre de stocker les informations des cellules à chaque itération. Cela permet au script de tourner sans envoyer les informations à chaque itération. Sans cela, le programme évaluerait chaque itération tandis que nous souhaitons que le programme n'évalue la grille d'une fois entièrement divisée.

12.2.3.3 - Texte d'information

```

#[0; n ] # ligne du haut
#[len-n;len] # Ligne du bas
#i%n=0 #multiples de n (colonne du côté droit)
#i-n-1%n=0 #colonne de gauche

```

Ce texte est informatif. Il est ici afin de rappeler comment identifier les cellules de chaque bord de la grille.

12.2.3.4 - Voisins à exclure

```
#Voisins à exclure pour les cellules des bords de la grille
LigneDuHaut=[-1-n,-n,-n+1]
LigneDuBas=[n-1,n,n+1]
ColonneGauche=[-n-1,-1,n-1]
ColonneDroite=[-n+1,1,n+1]
```

Cette partie nous permet de renseigner les voisins à exclure pour les cellules se trouvant sur le bord de la grille. Au lieu de préciser le voisinage à chaque opération, il suffit désormais de préciser l'un de ces quatre textes afin d'exclure les voisins indésirables.
La "LigneDuHaut" exclut les voisins se situant au-dessus à gauche, au-dessus et au-dessus à droite.
La "LigneDuBas" exclut les voisins se situant en dessous à gauche, en dessous et en dessous à droite.
La "ColonneGauche" exclut les voisins se situant au-dessus à gauche, à gauche et en dessous à gauche.
La "ColonneDroite" exclut les voisins se situant au-dessus à droite, à droite et en dessous à droite.

12.2.3.5 - Voisinage des cellules

```
#Voisins=[-n,-1,+1,n] #Tous les voisins d'une cellule, voisinnage de Von Neumann
Voisins=[-n,-1,+1,n,-n-1,-n+1,n-1,n+1] #Tous les voisins d'une cellule, voisinnage de Moore

```

La cinquième partie du script permet de sélectionner le voisinage souhaité. Pour cette recherche, j'ai utilisé le voisinage de Moore. Cependant, il est possible d'activer le voisinage de Von Neumann en retirant le "#" de la première ligne et en l'insérant au début de la seconde. Le "#" permet de préciser que nous écrivons du texte et que le programme ne doit pas le prendre en compte. Ce voisinage sera utilisé par toutes les cellules. Dans les cas particuliers qui sont les bords de la grille, il faudra exclure de ce voisinage les voisins indésirables précisés dans la partie 4.

12.2.3.6 - Boucle de répétition

```
def over():
    for i in range(0,len(C)):
        if(C[i] == 0) :
            return False
    return True

for z in range(50) :
    print(z)
    for i in range(0,len(C)):
        x[i]=0
        if C[i]==0: #and (N[i]>=1):
```

C'est dans cette sixième partie que l'algorithme commence à observer les cellules. Ce script observe dans l'entièreté de la grille s' il trouve une cellule ayant la valeur 0 (état éteint). Si tel est le cas, la fonction "over" est fausse. Le cas échéant la fonction "over" est vraie, cette boucle s'arrête et le programme passe à la suite.

Arthur ROULAND

12.2.3.7 - Etude des bords de la grille

```
    #Ligne du haut
    
            if i>=0 and i<n:
                for k in Voisins:
                    if i%n==0:
                        if k not in LigneDuHaut and k not in ColonneGauche:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    elif (i+1)%n==0:
                        if k not in LigneDuHaut and k not in ColonneDroite:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    else :
                        if k not in LigneDuHaut:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
    
    
    
    
    #Ligne du bas
    
            elif i>=len(C)-n-1 and i<len(C):
                for k in Voisins:
                    if i%n==0:
                        if k not in LigneDuBas and k not in ColonneGauche:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    elif (i+1)%n==0:
                        if k not in LigneDuBas and k not in ColonneDroite:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    else:
                        if k not in LigneDuBas:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
    
    
    
    
    #Colonne de gauche
    
            elif i%n==0:
                for k in Voisins:
                    if i>=0 and i<n:
                        if k not in ColonneGauche and k not in LigneduHaut:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    elif i>=len(C)-n-1 and i<len(C):
                        if k not in ColonneGauche and k not in LigneduBas:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    else:
                        if k not in ColonneGauche:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
    
    
    
    #Colonne de droite
    
            elif (i+1)%n==0:
                for k in Voisins:
                    if i>=0 and i<n:
                        if k not in ColonneDroite and k not in LigneduHaut:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    elif i>=len(C)-n-1 and i<len(C):
                        if k not in ColonneDroite and k not in LigneduBas:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
                    else:
                        if k not in ColonneDroite:
                            j=i+k
                            if C[j]!=0:
                                x[i]=(C[j])
```


Dans le cas où “over” est faux, notre algorithme va venir dans cette partie qui est divisée en 4 éléments presque identiques. La logique de ces 4 parties est la même, seules les formules de voisinages varient en fonction du bord étudié.
Prenons l'exemple de la ligne du haut. Dans ce cas, l'algorithme fonctionne de la manière suivante.
Si i (notre cellule étudiée) est plus grand ou égal à 0 et que i est inférieur à n (cela correspond à la ligne du haut) Dans ce cas, l'algorithme sait que la cellule se trouve donc sur la ligne du haut. Le cas échéant, il serait passé aux autres étapes). Maintenant, afin d'exclure les voisins, il est nécessaire de vérifier si notre cellule ne se situe pas dans un coin. Si elle se situe dans le coin en haut à gauche, nous allons exclure les voisins de LigneDuHaut et de ColonneGauche. Si elle se situe dans le coin en haut à droite, nous allons exclure les voisins de LigneDuHaut et de ColonneDroite. Autrement, nous n'allons exclure que les voisins se trouvant dans LigneDuHaut.
Cette opération est répétée pour la ligne du bas, la colonne de droite ainsi que la colonne de gauche.

12.2.3.8 - Etude des cellules génériques

```
            else :
                for k in Voisins:
                    j=i+k
                    if C[j]!=0:
                        x[i]=(C[j])
    
    
    
        else :
            x[i]=C[i]            #future valeur de la case i
    
```

Si la cellule étudiée ne rentre dans aucune des parties de la partie 7, cela équivaut à dire que cette cellule ne se situe pas sur un des bords de la grille. Dans ce cas, le voisinage utilisé correspond au voisinage de Moore au complet.

L'algorithme, dans la partie 7 et dans la partie 8 va regarder ses voisins. La seule différence est que dans la partie 7 les voisins ne sont pas au complet. En regardant ses voisins, l'algorithme dit que si notre cellule étudiée a au moins un voisin dont l'état est différent de zéro (cela signifie qu'il a une couleur autre que la noire), alors notre cellule étudiée va stocker en mémoire la valeur de ce voisin. En effet, la cellule ne va pas tout de suite changer d'état. Autrement, la prochaine cellule étudiée verra qu'elle a une couleur et cela fausserait le résultat.
Enfin, une fois que toutes les cellules de la grille auront été étudiées, toutes celles qui ont gardé une couleur en mémoire vont se voir affecter cette couleur.

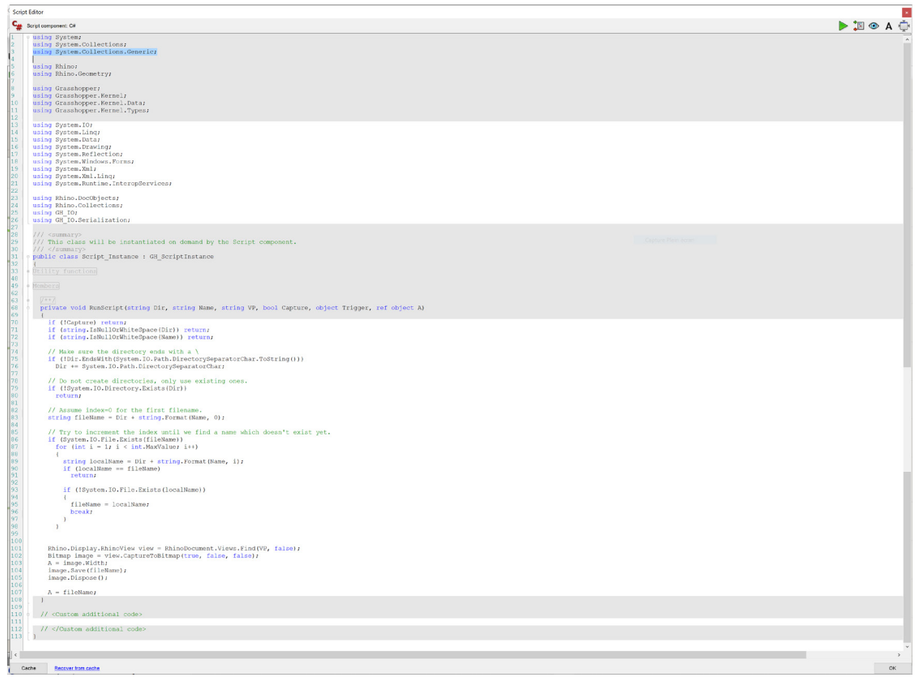
12.2.3.9 - Affectation des couleurs

```
    for d in range(0,len(C)):
        C[d] = x[d]
    
    if (over() == True) :
        break
    
print (len(C))
print (len(x))

for i in range(0,len(C)):
    a.append (x[i])

print(over())
```

Suite à cette opération, l'algorithme revient à la partie 6. Cette partie va regarder s' il reste une cellule à l'état 0. Si c'est le cas, alors le script va tourner à nouveau. C'est une itération. Si aucune cellule n'est à l'état 0, cela veut dire que chaque cellule a une couleur et donc que la division de la grille est terminée. La fonction “over” est vraie. L'algorithme dit donc que toutes les cellules se trouvant entre 0 et la longueur de la grille (soit toutes les cellules de la grille), vont stocker leur valeur respective dans le paramètre “a”. Pour rappel, “a” est la sortie du composant python dans grasshopper, cela signifie que les valeurs sont envoyées dans la suite de l'algorithme.

Arthur ROULAND

12.3 - Capture d'écran de l'enregistreur



12.3.2 - Code de l'enregistreur

```
using System;
using System.Collections;
using System.Collections.Generic;

using Rhino;
using Rhino.Geometry;

using Grasshopper;
using Grasshopper.Kernel;
using Grasshopper.Kernel.Data;
using Grasshopper.Kernel.Types;

using System.IO;
using System.Linq;
using System.Data;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using System.Xml;
using System.Xml.Linq;
using System.Runtime.InteropServices;

using Rhino.DocObjects;
using Rhino.Collections;
using GH_IO;
using GH_IO.Serialization;

/// <summary>
/// This class will be instantiated on demand by the Script component.
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
#region Utility functions
  /// <summary>Print a String to the [Out] Parameter of the Script component.</summary>
  /// <param name="text">String to print.</param>
  private void Print(string text) { /* Implementation hidden. */ }
  /// <summary>Print a formatted String to the [Out] Parameter of the Script component.</summary>
  /// <param name="format">String format.</param>
  /// <param name="args">Formatting parameters.</param>
  private void Print(string format, params object[] args) { /* Implementation hidden. */ }
  /// <summary>Print useful information about an object instance to the [Out] Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj) { /* Implementation hidden. */ }
  /// <summary>Print the signatures of all the overloads of a specific method to the [Out] Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj, string method_name) { /* Implementation hidden. */ }
#endregion

#region Members
  /// <summary>Gets the current Rhino document.</summary>
  private readonly RhinoDoc RhinoDocument;
  /// <summary>Gets the Grasshopper document that owns this script.</summary>
  private readonly GH_Document GrasshopperDocument;
  /// <summary>Gets the Grasshopper script component that owns this script.</summary>
  private readonly IGH_Component Component;
  /// <summary>
  /// Gets the current iteration count. The first call to RunScript() is associated with Iteration==0.
  /// Any subsequent call within the same solution will increment the Iteration count.
  /// </summary>
  private readonly int Iteration;
#endregion

  /// <summary>
  /// This procedure contains the user code. Input parameters are provided as regular arguments,
  /// Output parameters as ref arguments. You don't have to assign output parameters,
  /// they will have a default value.
  /// </summary>
  private void RunScript(string Dir, string Name, string VP, bool Capture, object Trigger, ref object A)
  {
    if (!Capture) return;
    if (string.IsNullOrWhiteSpace(Dir)) return;
    if (string.IsNullOrWhiteSpace(Name)) return;

    // Make sure the directory ends with a ¥
    if (!Dir.EndsWith(System.IO.Path.DirectorySeparatorChar.ToString()))
      Dir += System.IO.Path.DirectorySeparatorChar;

    // Do not create directories, only use existing ones.
    if (!System.IO.Directory.Exists(Dir))
      return;

    // Assume index=0 for the first filename.
    string fileName = Dir + string.Format(Name, 0);

    // Try to increment the index until we find a name which doesn't exist yet.
    if (System.IO.File.Exists(fileName))
      for (int i = 1; i < int.MaxValue; i++)
      {
        string localName = Dir + string.Format(Name, i);
        if (localName == fileName)
          return;

        if (!System.IO.File.Exists(localName))
        {
          fileName = localName;
          break;
        }
      }


    Rhino.Display.RhinoView view = RhinoDocument.Views.Find(VP, false);
    Bitmap image = view.CaptureToBitmap(true, false, false);
    A = image.Width;
    image.Save(fileName);
    image.Dispose();

    A = fileName;
  }

  // <Custom additional code>

  // </Custom additional code>
}
```

Concevoir et construire l'architecture

Activités et Instrumentation de la conception

Arthur ROULAND

2021

Encadré par

François Guéna
Joaquim Silvestre
Anne Tüscher