RAPPORT DE STAGE

« Rendu graphique des matériaux mouillés avec Substance Designer »

Juin – Juillet 2019

Loïc LACHIVER
Référent université : Carole BLANC
Tuteurs : Pascal BARLA et Gaël GUENNEBAUD

INRIA – Bordeaux Sud-Ouest
Université de Bordeaux – Promotion 2018-2019

Sommaire

Remerciements

Je remercie Carole BLANC, pour m'avoir conseillé de postuler pour une offre de stage au sein de l'équipe MANAO de l'INRIA,

Pierre BENARD, pour m'avoir accueilli au sein de l'équipe,

Pascal BARLA et Gaël GUENNEBAUD, pour m'avoir transmis une partie de leurs connaissances et accompagné tout au long du stage,

L'université de Bordeaux, pour m'avoir offert l'opportunité de faire un stage m'ayant aidé à mieux définir mon orientation professionnelle.

Résumé / Abstract

Dans le cadre de ma licence informatique à l'université de Bordeaux, j'ai eu l'opportunité de faire un stage de deux mois (du 3 juin au 31 juillet 2019) à l'Institut national de recherche en informatique et en automatique (Inria), au sein du centre de recherche « Bordeaux Sud-Ouest » basé sur le campus de Talence.

Mon sujet de stage étant le « rendu graphique de matériaux mouillés », j'ai intégré l'équipe MANAO dont la recherche porte sur l'interaction entre lumière, forme et matière. J'ai été amené à explorer la physique de la lumière et des matériaux, j'ai appris à maîtriser le logiciel Substance Designer et j'ai également découvert le shading sous OpenGL. J'ai pu créer un graphe Substance capable de rendre mouillé n'importe quel matériau suivant certains paramètres. L'objectif était d'offrir aux artistes un outil simple pour mouiller un matériau, mon travail jonglait donc entre graphisme, physique et informatique.

Dans ce rapport, je présenterai mon travail durant ces deux mois de stage en abordant quelques notions d'image de synthèse, en définissant la problématique et en présentant mon graphe Substance et son shader associé.

As part of my studies in the informatics licence of the University of Bordeaux, I had the chance to do a 2 months traineeship (from the 3rd of June to the 31st of July 2019) in the National Institute for Research in Computer Science and Automation (Inria), within the « Bordeaux Sud-Ouest » research center, based in the campus of Talence.

My subject of traineeship being « graphic rendering of wet materials », I joined the « MANAO » team, whose research is based on the interaction between light, shape and matter. I was led to explore physics of light and materials, I learned to master the software Substance Designer et I also discovered the shading under OpenGL. I have created a Substance graph able to wet any material following some parameters. The goal was to give to artists a simple tool to wet a material, thus my work juggled with graphism, physics and informatics.

In this report, I will present my work during this 2 month traineeship by adressing some notions about image synthesis, by defining the problematic and by presenting my graph and its associated shader.

I. Introduction

L'Institut national de recherche en informatique et en automatique (Inria) est un établissement public à caractère scientifique et technologique français spécialisé en mathématiques et informatique, fondé le 3 janvier 1967. Il est présidé par Bruno Sportisse et compte un effectif de 2500 personnes. Le budget de l'Inria en 2008 était de 236 000 000 euro.

Depuis 2008, l'Inria possède 8 centres de recherche autonomes dont celui nommé « Bordeaux Sud-Ouest », situé à Talence. Au sein de celui-ci travaille l'équipe « MANAO » qui étudie comment la lumière, la matière et les formes interagissent en synergie (site web : manao.inria.fr). C'est dans cette équipe que j'ai effectué mes deux mois de stage.

Ayant obtenu un DUT GEII en 2018 (GEII : Génie Electrique et Informatique Industrielle, bien que ce soit centré sur l'électronique), j'ai choisi de poursuivre mes études en l'informatique, car c'est un domaine qui me passionne depuis l'enfance et pour lequel j'ai développé une intuitivité.

Je me sens particulièrement attiré par l'informatique graphique, effectuer un stage au sein de l'équipe MANAO de l'Inria était donc une confirmation de mon souhait d'orientation professionnelle. Avant ma postulation pour le stage, j'avais parcouru les publications accessibles sur le site de MANAO, et la pluralité des domaines où s'applique l'image de synthèse m'avait ouvert les yeux sur un pan de l'informatique que je connaissais peu. Animation, architecture, culture (musées etc.)… autant de domaines qui me passionnent également et pour lesquels j'aimerais apporter ma contribution par le biais de l'informatique.

II. Contexte

a. La lumière en image de synthèse

Depuis les premiers long-métrages en image de synthèse, la recherche n'a cessé d'inventer de nouvelles techniques de rendu 3D afin de modéliser au mieux la physique de la lumière. Le livre « Physically Based Rendering », sorti en 2004 et écrit par Matt Pharr et Wenzel Jakob, formalise ces techniques qui se démocratiseront par la suite dans le milieu de l'infographie, créant ainsi le « rendu physiquement réaliste » abrégé en « PBR ». Pour qu'un rendu soit considéré comme « PBR », il faut qu'il respecte deux modèles physiques :

Premièrement, la théorie des micro-facettes. Elle décrit la structure microscopique d'un matériau comme un ensemble de petites facettes se comportant comme des miroirs. Lorsqu'elles sont alignées, on observera uniquement des reflets dits « spéculaires ». A l'inverse, lorsque les micro-facettes sont désordonnées, les reflets seront diffus.

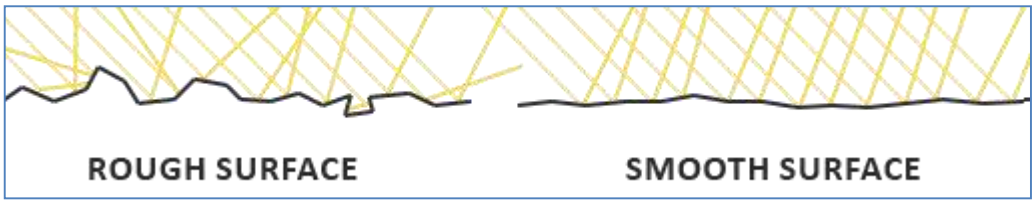


Figure 1 : réflexion de la lumière à la surface d'un matériau suivant la théorie des micro-facettes

En infographie, l'alignement des micro-facettes est défini par la « rugosité » d'un matériau, qui peut avoir une valeur de 0 à 1. Une rugosité de 0 correspond à des reflets spéculaires maximum, une rugosité de 1 à une réflexion totalement diffuse.

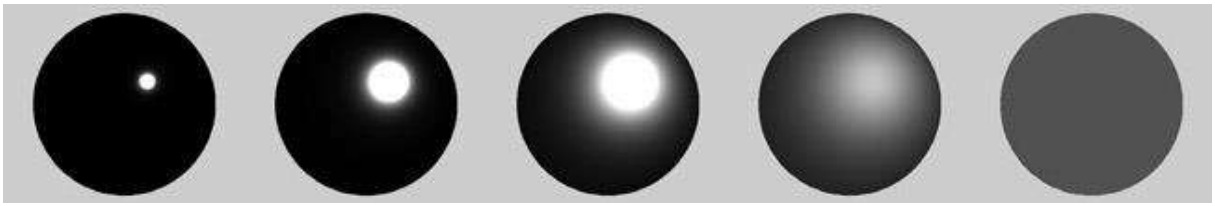


Figure 2 : de gauche à droite : rugosité de 0.1, 0.3, 0.6, 0.8 et 1.0

Deuxièmement, le principe de conservation de l'énergie doit être respecté. La quantité de lumière réfléchie doit être égale à la quantité de lumière incidente. La relation entre lumière réfléchie et incidente est décrite par la « BRDF » (Bidirectionnal Reflective Distribution Function). Cette fonction est définie dans le shader.

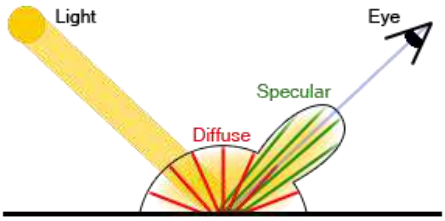


Figure 3 : illustration d'une BRDF : on voit que la lumière réfléchie est la somme des reflets diffus et spéculaires

b. Les matériaux

Dans les premiers jeux-vidéos, les matériaux comportaient une unique texture de couleur. Maintenant, un matériau est décrit par plusieurs textures (appelées maps) décrivant le comportement de la lumière à sa surface.

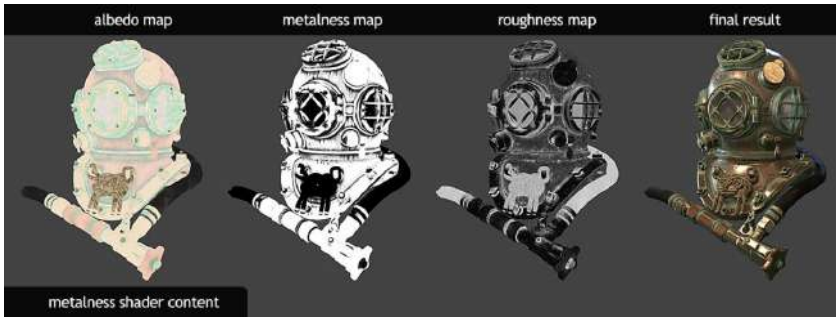


Figure 4 : les maps principales d'un rendu PBR : albedo, metalness et roughness

Prenons l'exemple de la figure 4. La map d'albedo correspond à la couleur du matériau. La map « metalness » décrit les parties métalliques en blanc, la couleur du métal sera celle de l'albedo. Enfin, la rugosité (roughness) du matériau est définie par une map en niveau de gris (noir -> pas rugueux, blanc -> rugueux). La combinaison de ces 3 maps, indispensables à un rendu PBR, donne le résultat visible à droite sur la figure 4.

Le workflow de maps le plus répandu pour la définition d'un matériau est le workflow « metallic / roughness ». Il comprend quelques maps en plus, et est largement utilisé en infographie 3D (animation, jeux vidéo).

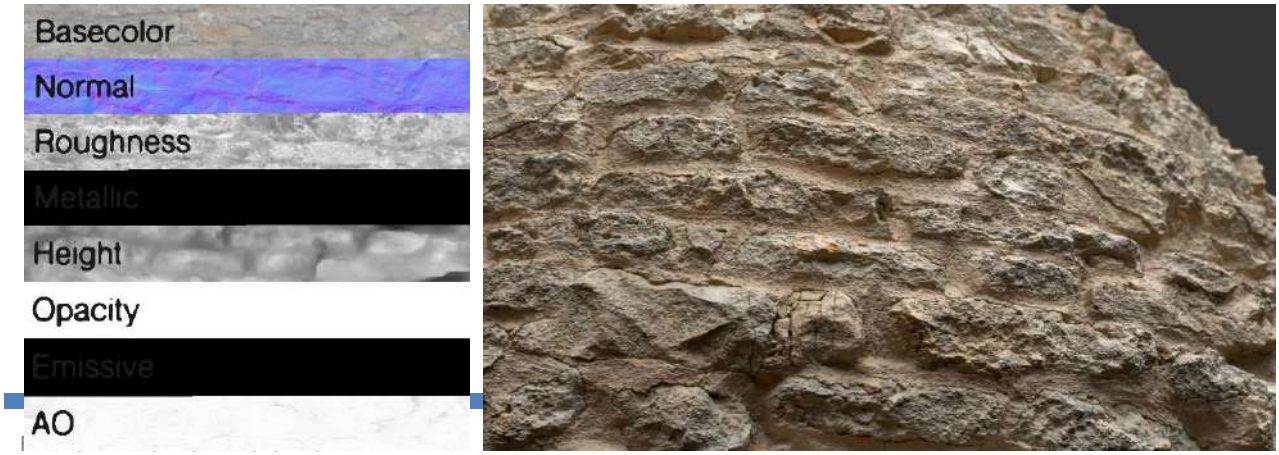


Figure 5 : les maps du workflow « metallic / roughness » à gauche, et le rendu du matériau à droite

La normal map change la rotation du vecteur de normale du matériau, ce qui changera la direction de réflexion de la lumière. Par défaut, le vecteur normal a pour coordonnées (0,0,1), ainsi un rayon lumineux frappant le matériau à la verticale aura une direction de réflexion également verticale. On peut donc donner une sensation de relief au matériau en modifiant les coordonnées du vecteur normal. Les valeurs x/y/z du vecteur seront représentées par les valeurs rouge/vert/bleu de la normal map, et « re-mappées » de [-1 ; 1] à [0 ; 1].

La height map donne une information de hauteur. Le shader pourra représenter le relief soit par la parallaxe, soit par la tessellation. On définiera les différentes techniques dans la partie « shader » du rapport.

c. Substance Designer

Au cours de ces deux mois de stage, j'ai utilisé le logiciel Substance Designer, développé par la société Allegorithmic qui a récemment été acquise par Adobe. Ce programme, largement utilisé dans l'industrie du jeu vidéo, est destiné à la génération de maps pour matériaux PBR à partir d'un processus de calcul représenté par un graphe éditable, que l'on appelle une « substance ». Ainsi, la génération des maps est totalement paramétrable et légère, car il n'y a pas d'images stockées dans la substance.

Chaque graphe est composé de nœuds contenant des entrées et des sorties, et un nœud peut également être un sous-graphe.

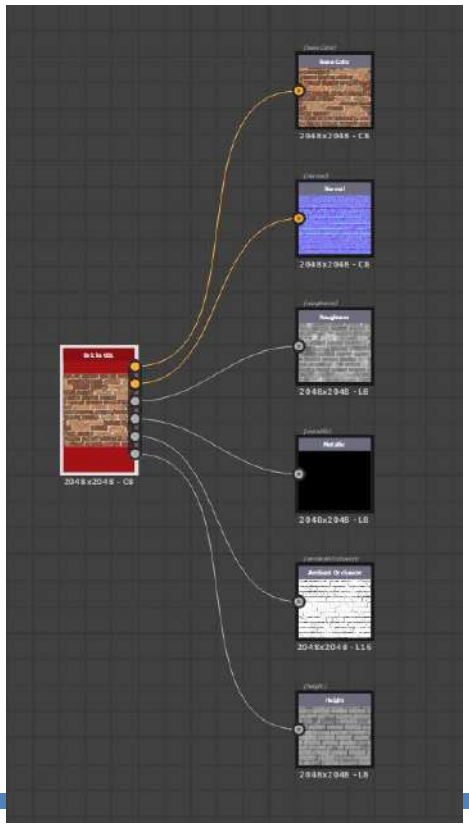
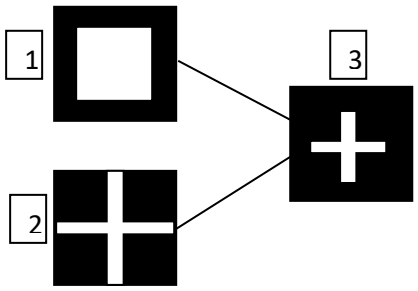


Figure 6 : Graphe de génération d'un matériau PBR de briques

Le graphe de la figure 6 comporte à gauche un nœud générant un matériau PBR de brique. On observe les maps du workflow metallic/roughness en sortie, à droite.

Le nœud rouge est un sous-graphe générant le matériau « brique » via une chaîne de compositing. Prenons un exemple de compositing simple :



Voici un graphe substance simplifié. Nous avons deux images générées dans les nœuds 1 et 2. Le nœud 3 est appelé « blend », il calcule le résultat d'une opération entre deux images, ici la multiplication.

La génération des briques est donc un processus de compositing et d'applications d'effets (flou, distorsion, damier etc.) qui calcule chacune des maps du workflow metallic/roughness. L'avantage de cette technique est que la génération de matériau ne se base pas sur des photos. Le graphe est donc léger, et la génération totalement paramétrable (on peut choisir par exemple le nombre de briques, leurs couleurs etc.). De plus, on peut rendre aléatoire certains paramètres de génération. Ainsi, si l'on souhaite modéliser un immense mur de brique, on peut éviter l'effet « damier de textures » et faire en sorte qu'il n'y ait pas une seule fois le même matériau.

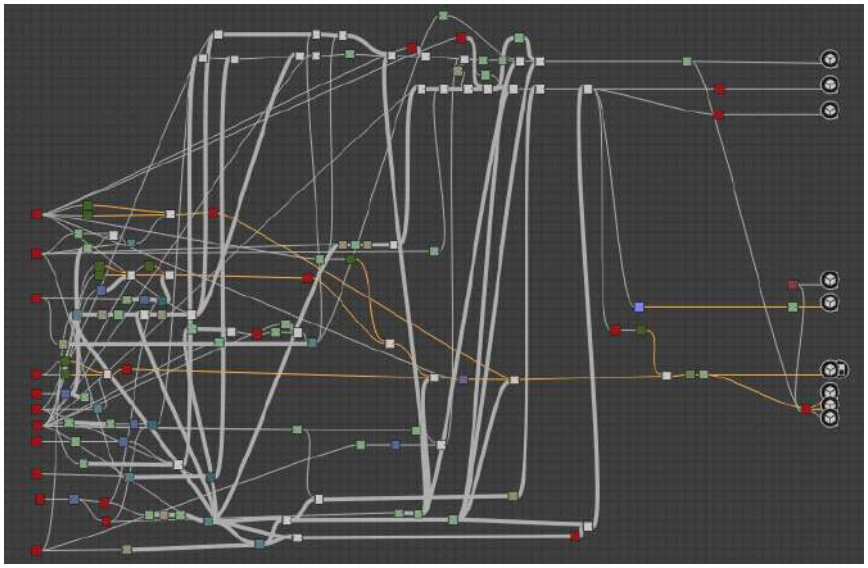


Figure 7 : le graphe complet de génération des briques

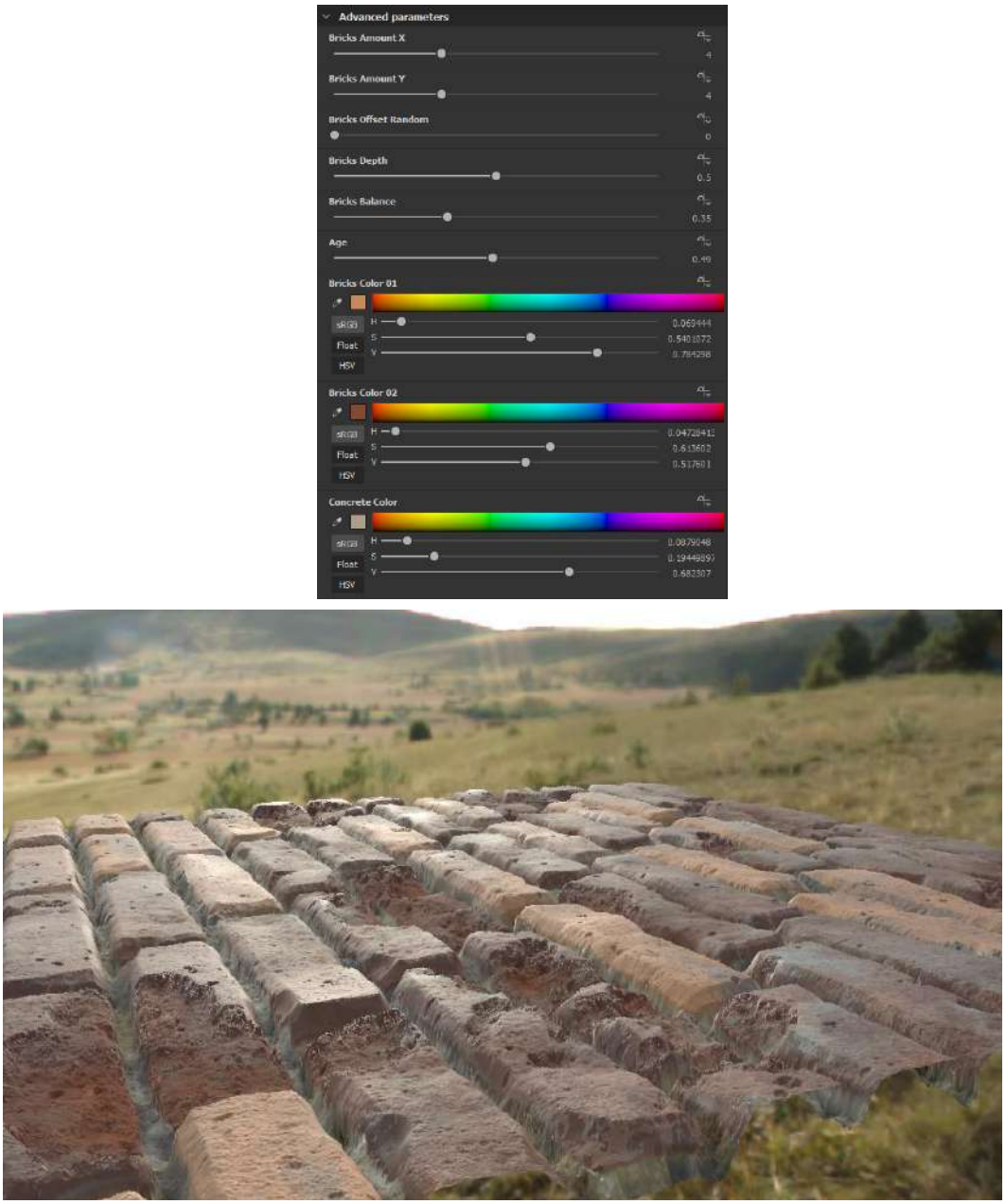


Figure 8 : paramètres de génération des briques (en haut) et rendu du matériau sous OpenGL (en bas)

Concernant le rendu 3D, Substance comprend deux moteurs : **openGL** et **iRay**. OpenGL utilise la **rastérisation**, le rendu se base uniquement sur des **calculs géométriques** définissant les polygones perçus par la caméra. iRay se base sur une technique plus gourmande nommée le lancer de rayon (**raytracing** en anglais) : des photons virtuels sont lancés depuis la caméra et leurs trajectoires et rebonds définissent ainsi les ombres, réflexions et réfractions non calculées par la rastérisation.

L'image qu'on voit derrière les briques en figure 8 est la map d'environnement qui définit l'éclairage de la scène. Le fait d'utiliser une image en tant que lumière de la scène s'appelle le « **Image Based Lighting** ».

d. Les matériaux mouillés

Revenons à la réalité pour un instant. Lorsqu'on compare un trottoir sec à un trottoir mouillé, on observe que le matériau devient plus sombre et des reflets spéculaires apparaissent.

Figure 9 : Trottoirs secs et mouillés (source : seblagarde.wordpress.com)

Pour comprendre pourquoi, il faut analyser le parcours de la lumière à la surface d'un matériau.

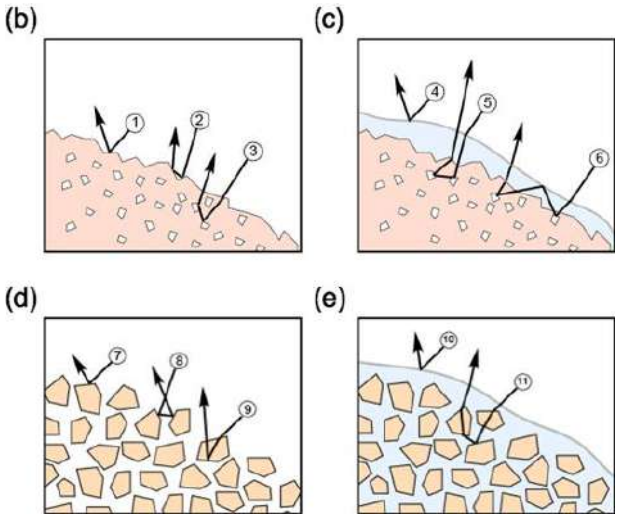


Figure 10 : trajets de la lumière à la surface de matériaux non poreux (a et b) et poreux (d et e). source : jov.arvojournals.org

Sur ce schéma, on observe les différents trajets de la lumière à la surface d'un matériau peu poreux (b et c) et d'un matériau poreux (d et e). La lumière qui traverse le matériau pour en ressortir (9) est responsable de la couleur de celui-ci. L'indice de réfraction du matériau étant plus proche de celui de l'eau que de l'air, la lumière passant à travers la couche d'eau aura plus de chances de continuer son chemin dans la même direction plutôt que d'être réfléchie (on voit que la lumière parcourt un chemin plus long en 11 qu'en 9). L'absorption est donc plus importante, par conséquent le matériau nous apparaît plus sombre et plus saturé.

De plus, la couche d'eau ayant une rugosité nulle, elle ne renverra que des reflets spéculaires (4 et 10).

Lorsqu'un matériau est recouvert d'une couche d'eau, on observe un autre effet : les ombres crées par la forme du matériau sont atténuées en présence d'une couche d'eau.

Le schéma ci-dessus représente le trajet de la lumière pour un matériau sec (gauche) et mouillé (droite). L'ombre 2 est plus petite que l'ombre 1 à cause de la réfraction.

Figure 11 : illustration du schéma précédent : les ombres sont atténuées lorsque le matériau est mouillé

Enfin, un autre phénomène physique rarement représenté dans les rendus actuels est la tension superficielle, qui créé les ménisques. La taille de ceux-ci donne une apparence plus ou moins visqueuse au liquide.

L'objectif de mon stage est donc celui-ci : **à partir d'un matériau PBR existant, représenter ces phénomènes physiques en modifiant les maps du matériau pour lui donner une impression de mouillé.**

Sur Substance Designer, il existe déjà un nœud nommé « water level ». C'est un nœud natif du logiciel qui prend en entrée les maps d'un matériau PBR et ressort le même matériau présentant un aspect mouillé. Il y a plusieurs paramètres dans ce nœud :

- Water level (de 0 à 1) : contrôle la hauteur de la surface de l'eau comprise entre la hauteur minimum et maximum de la height map du matériau. A 0, le matériau est sec et à 1 il est totalement immergé.

- Wet distance (de 0 à 1) : contrôle la hauteur des bords de la partie mouillée.

- Edges wetness (de 0 à 1) : définit la rugosité de la partie mouillée.

- Autres paramètres : water darkness, sludge color & sludge depth (contrôle la présence de boue dans l'eau), frost (niveau de gel de l'eau) etc.

Voici le rendu d'un matériau brique auquel on a ajouté le nœud « water level » :

Figure 12 : exemple de rendu du noeud "water level"

On remarque plusieurs problèmes sur la figure 12.

Il n'y a pas de partie "humide" (où l'eau a déjà été absorbée par le matériau), car il n'y a que deux zones. Les paramètres de contrôle de zone sont d'ailleurs peu précis, car tout est contrôlé par la hauteur. On ne peut pas contrôler la taille de la zone mouillée avec la distance par rapport aux bords de la partie immergée.

Visuellement, le résultat est peu convaincant (surtout de près). Les bords de la partie immergée sont trop nets car la tension de surface n'est pas prise en compte. La partie mouillée est peu réaliste car le nœud se contente d'uniquement diminuer la rugosité, ce qui amplifie les défauts du matériau. De plus, les pentes abruptes du matériau (exemple : les bords des briques) sont également mouillés, or la physique voudrait que l'eau ne tienne pas sur les parois.

Enfin, la partie immergée **écrase** la hauteur, la normale et la rugosité existantes du matériau. Cela est dû au fait qu'un matériau PBR ne peut **pas** avoir plusieurs couches, ses maps définissent une seule information par **texel**. Etant donné qu'on perd le relief original du matériau, les phénomènes de réfraction ne sont pas non plus représentés.

III. Notre approche

a. Identification des zones

La problématique du stage était donc d'améliorer le nœud « water level ». Avec Pascal, nous sommes partis du constat qu'il n'existait non pas deux zones comme définies dans le nœud original, mais trois zones :

- La zone immergée (**immersed**), la surface de l'eau est totalement horizontale, décorrélée de celle du matériau. Cette zone existe déjà dans le nœud water level.

- La zone mouillée (**wet**) : une fine couche d'eau épouse la surface du matériau, il y a donc toujours des reflets spéculaires. Cette zone existe aussi dans le nœud water level, mais nous chercherons à prendre plus de phénomènes physiques en compte. Nous avons observé avec Pascal que la transition entre immergé et mouillé n'est pas brutale : en plus des ménisques, des petites flaques se présentent au bord de la partie immergée et leur nombre décroît à mesure qu'on s'éloigne des bords.

- La zone humide (**moist**), qui représentera la zone où l'eau a déjà été absorbée, ainsi il n'y a plus de reflets spéculaires. Cette zone n'est pas toujours présente, elle dépend de la porosité du matériau (ex : le plastique et le métal n'absorbent quasiment pas, contrairement au bois ou au carton).

Voici un tableau récapitulatif des zones et des transitions entre celles-ci, avec leur impact sur chacune des maps du matériau.

Zone / Map	Immersed	*Transition*	Wet	*Transition*	Moist
Albedo (couleur)	Matériau plus foncé et plus saturé				
Roughness	environ 0	dégradé	Plus faible que l'existant & roughness quasi-nulle pour les petites flaques	dégradé	existant
Height	Valeur du paramètre « water_level »	/	existant	/	existant
Normal	Uniforme vers le haut	Ménisque	Existant et uniforme vers le haut pour les petites flaques	/	existant
Ambient Occlusion	Plus claire	/	existant	/	existant

b. Mon graphe Substance

1. Création des zones

Sur mon graphe Substance, je souhaite tout d'abord générer **séparément** les **masques** correspondant à ces zones afin de pouvoir les exporter si je le souhaite. Il y aura donc un nœud générant les zones, et un nœud modifiant les maps du matériau suivant ces zones. Voici un schéma simplifié de mon graphe :

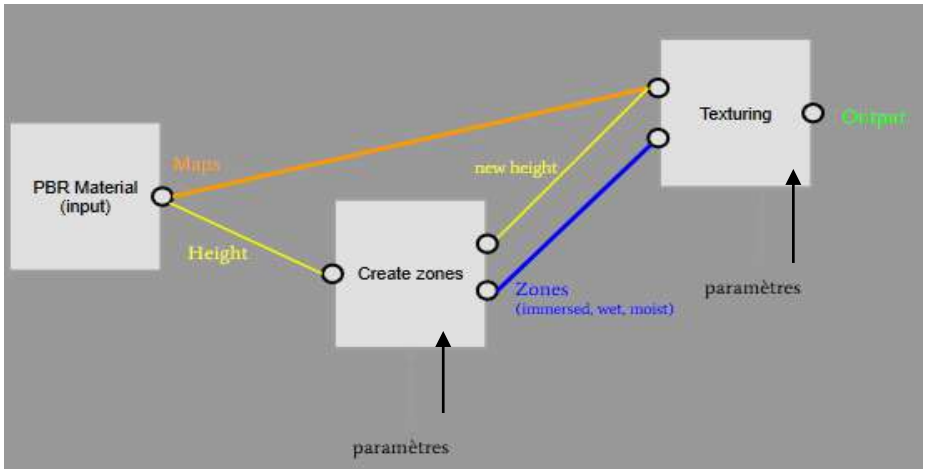


Figure 13 : processus de mon graphe Substance

On observe que la génération des zones se base uniquement sur la height map du matériau, comme dans le water level original. La création des zones prend plusieurs paramètres en entrée :

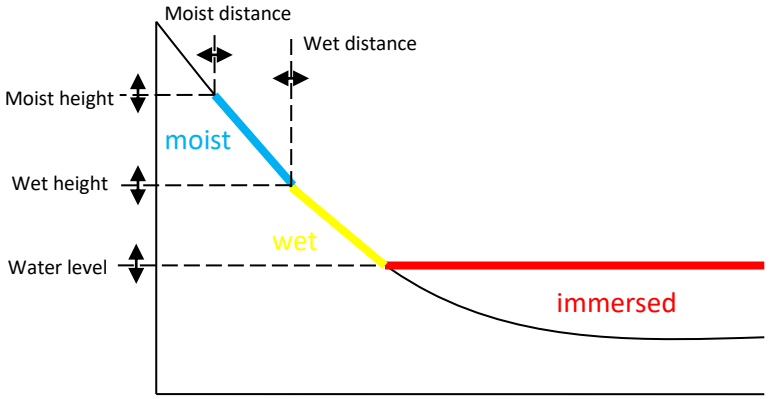
- water level (float) : hauteur de la surface de l'eau (de 0 à 1 : 0 -> sec, 1 -> totalement immergé)

- wet height : hauteur de la zone wet (de 0 à 1). Équivalent à « wet distance » du water level originel.

- wet distance : largeur de la zone wet (par rapport aux bords de la partie immergée).

- moist height & moist distance : même principe que pour wet height & wet distance.

- noise intensity : sert à déformer les bords de la partie wet à partir d'un bruit afin d'avoir des transitions plus réalistes entre wet et moist.



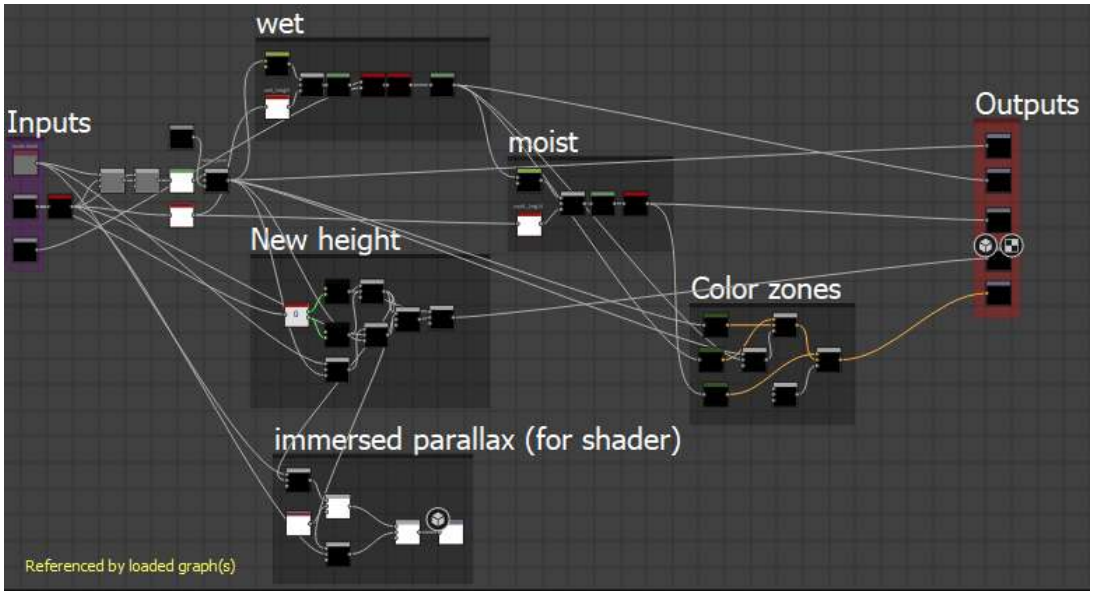


Figure 14 : le graphe de création de masques des différentes zones

Le sous-graphe « create masks » ((visible en grand en **annexe 1**) a 3 entrées (inputs) visibles à gauche de la figure 14 :

- le water level, représenté par un niveau de gris (0 -> noir, 1 -> blanc).
- La heightMap du matériau PBR.
- Le bruit de déformation, que l'artiste pourra changer à sa guise.

Suivant les paramètres définis plus haut, le graphe va calculer les masques correspondant aux zones immersed, wet et moist. Si on garde l'exemple des briques, voici les sorties du graphe « create masks » pour des paramètres choisis arbitrairement :

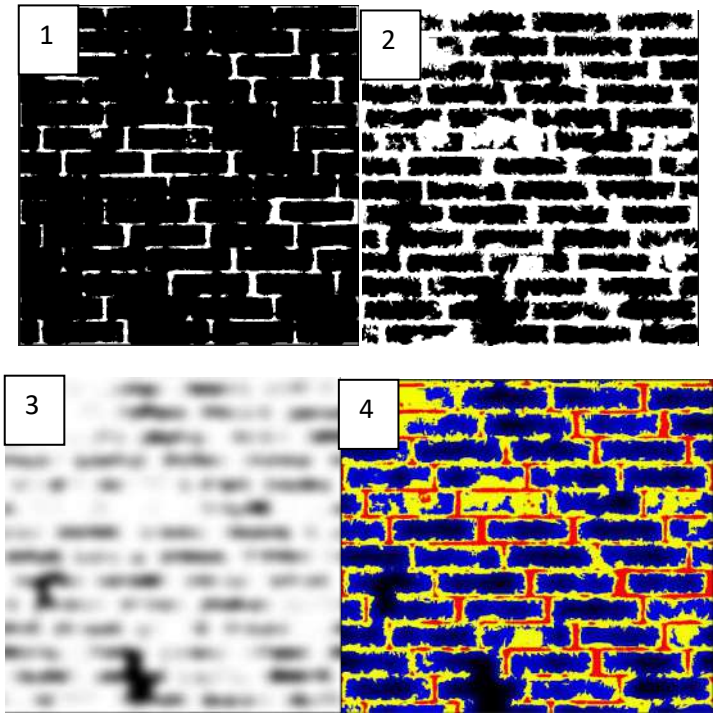


Figure 15 : pour un matériau brique > 1 : immersed zone, 2 : wet zone, 3 : moist zone, 4 : visualisation des zones

On remarque que les bords de la partie moist sont flous. En effet, dans la réalité, l'absorption de l'eau par le matériau ne s'arrête pas à une limite bien définie. De plus, mon nœud exporte une visualisation des zones en rouge / jaune / bleu qui pourra remplacer l'albedo du matériau afin de simplifier le contrôle des zones.

A partir du masque de la zone immergée, on vient modifier la heightMap du matériau :

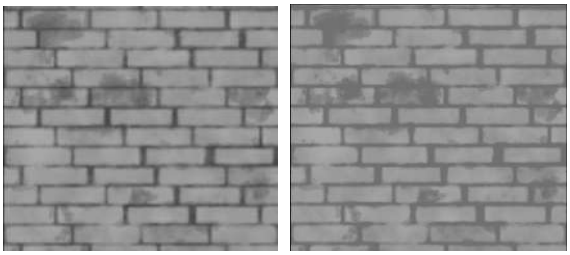


Figure 16 : la heightMap avant/après modification. On remarque la hauteur correspondante à la zone immergée est plus claire, car la surface de l'eau écrase la hauteur existante.

Un des autres avantages de la séparation de la création des zones et du texturing est que l'artiste peut **utiliser ses propres zones**, sans passer par le graphe « create masks ». Ainsi, il peut mouiller spécifiquement certaines parties du matériau.

2. Texturing

Maintenant que les zones ont été calculées, nous pouvons commencer à modifier les maps du matériau. Voici le graphe complet du nœud texturing (visible en grand en **annexe 2**).

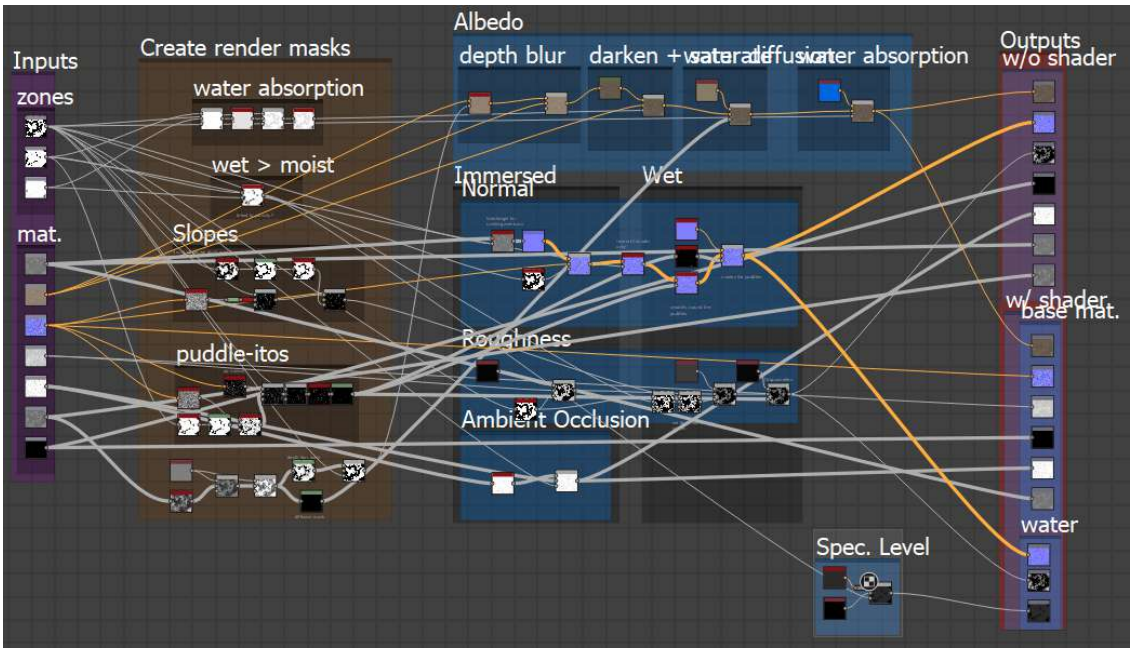


Figure 17 : le graphe de texturing (modification des maps du matériau PBR)

L'organisation du graphe permet de savoir d'un coup d'œil quelles maps sont modifiées. Concentrons-nous par exemple sur la modification de la normal map :

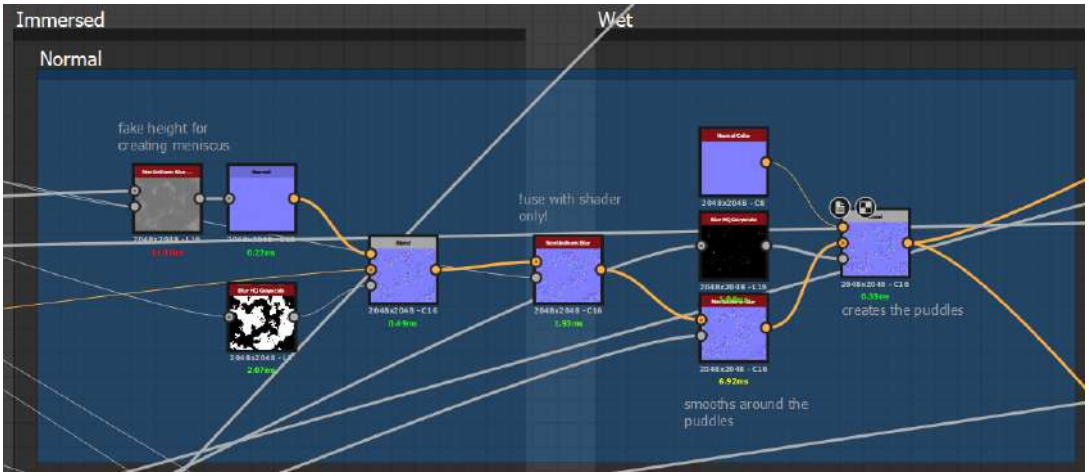


Figure 18 : compositing de la normal map

Afin de créer les ménisques, je créé une heightMap temporaire où je floute les bords de la partie immergée. Cela me permet d'avoir une transition entre la hauteur de l'eau et celle du matériau. Je transforme ensuite cette heightMap en normalMap, et j'ajoute les normal map des petites flaques.

Voici les paramètres pour le graphe « texturing » :

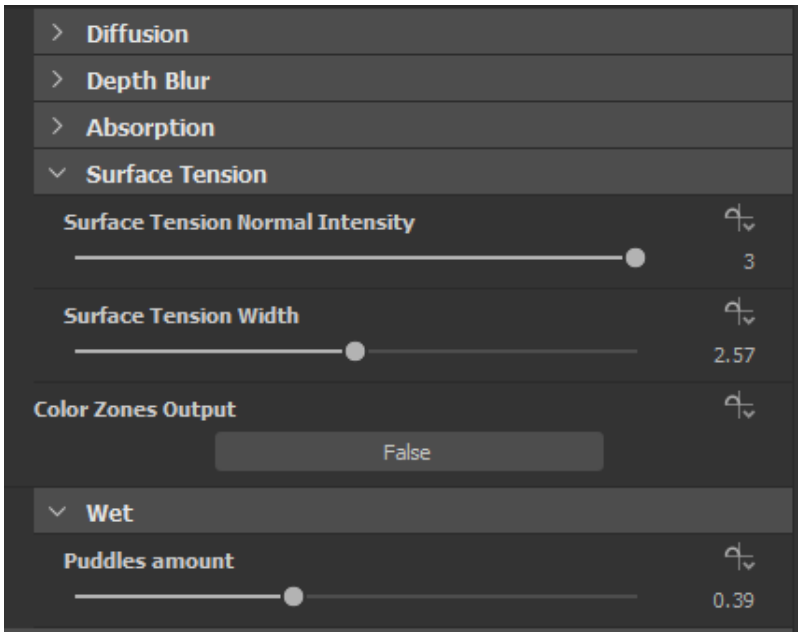
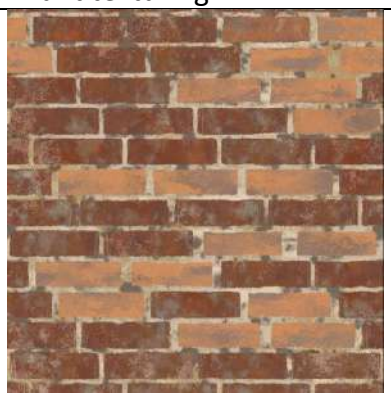
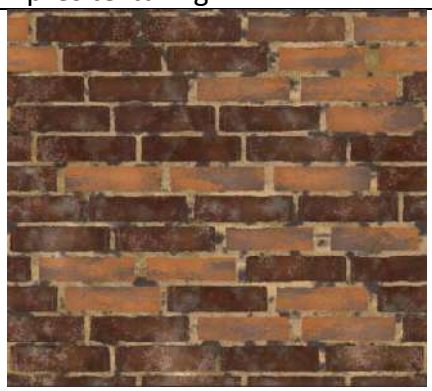
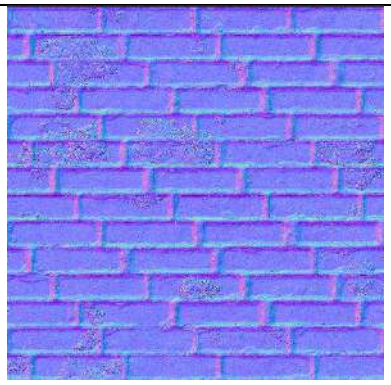
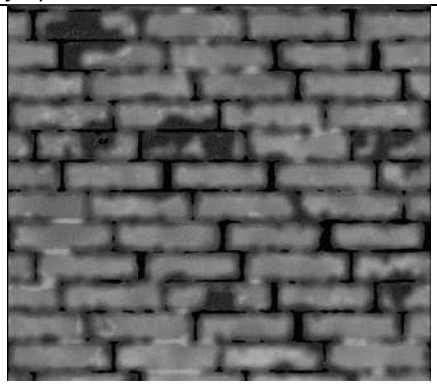
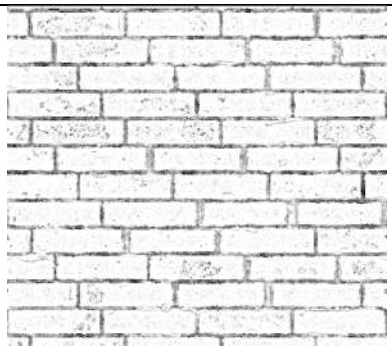
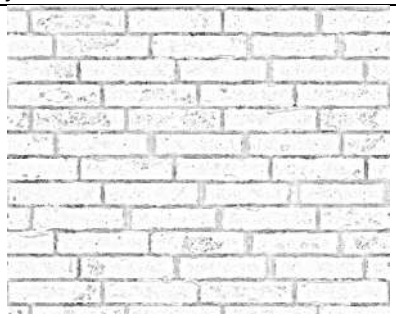


Figure 19 : paramètres du graphe "texturing"

Les paramètres du nœud texturing sont donc purement esthétiques et n'ont aucun effet sur les zones.

Etant donné que le détail du compositing de chaque map serait trop long, voici un tableau qui résume la modification des maps apportée par le nœud « texturing » :

	Avant texturing	Après texturing
Albedo		 *Le matériau est plus sombre est plus saturé*
Normal		 *La différence est difficile à voir, mais la normale est uniforme dans la zone immergée et dans les petites flaques.*
Roughness		 *La zone immergée et les petites flaques ont une rugosité quasi-nulle et la zone wet a une rugosité faible.*
Ambient Occlusion		 *L'occlusion ambiante est plus claire dans la zone immergée*

Voici un comparatif entre le rendu du water level et de mon nœud :

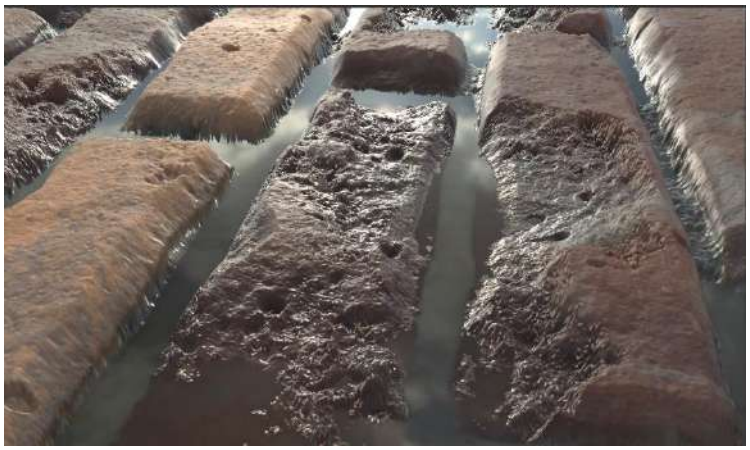


Figure 20 : rendu du matériau brique avec le "water level"

Figure 21 : rendu du matériau brique avec mon graphe Substance

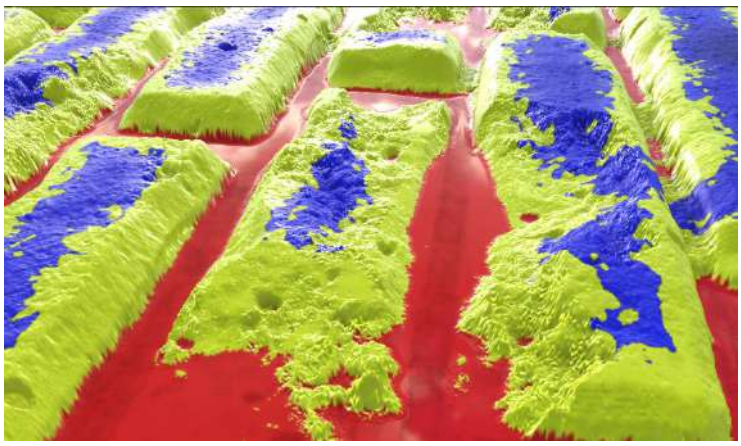


Figure 22 : visualisation des zones (rouge : immergé, jaune : mouillé, bleu : humide)

D’autres rendus sont présentés en annexe 3.

Pour rappel, je n’ai pas créé les matériaux PBR. Je les rendus mouillés avec mon graphe Substance.

3. Conclusion du texturing

Nous nous sommes rendu compte de l'importance des ménisques pour notre perception d'un matériau mouillé. Les transitions entre les zones sont beaucoup plus réalistes, notamment grâce aux petites flaques dans la transition immersed / wet. Le fait de ne pas mouiller les parois du matériau évite l'impression de viscosité et permet de mieux ressentir l'effet de la gravité sur l'écoulement de l'eau.

Seulement il subsiste toujours un problème majeur : l'écrasement du matériau par la zone immergée, ce qui nous fait toujours perdre les informations de height, roughness et normal du matériau sous la surface de l'eau.

Pour remédier à ce problème, nous avons imaginé trois solutions. Je détaillerai succinctement les deux premières car elles n'ont pas été adoptées :

1- Utilisation de deux meshes 3D

La première se rapproche de ce qui se fait dans le jeu vidéo. Au lieu d'appliquer le matériau à un seul **mesh**, on utilise un autre plan sur lequel on applique un matériau « eau », et ce plan viendra s'encastrer et se déplacer verticalement dans le mesh de base suivant le niveau de l'eau. Un avantage non négligeable est que le moteur de rendu iRay permet d'importer une autre map appelée « refraction », qui définit l'indice de réfraction du matériau.

Figure 23 : rendu de deux meshes 3D (matériau carrelage et matériau eau) avec prise en compte de la réfraction

Le problème est que cette solution n'est pas très portative car elle nécessite une scène 3D avec deux meshes. De plus, on souhaite garder un seul matériau alors qu'ici nous en utilisons deux.

2- MDL

Le MDL (Material Definition Language) est un langage de NVIDIA pour la définition de matériaux. Il permet de définir des matériaux multicouches, ce qui paraît intéressant au vu de ce que l'on souhaite faire. Substance Designer permet de créer des graphes MDL dont le rendu sera nécessairement fait sous iRay. Problème, nous nous sommes rendu compte qu'il n'est pas possible d'avoir plusieurs heightmap/normalMap par matériau, mais seulement plusieurs roughness. Ainsi nous retombons sur le même problème.

3- Création d'un shader personnalisé

Le shader est un programme executé en parallèle sur le GPU destiné à calculer la couleur de chaque pixel du rendu.

Nous avons constaté que l'on pouvait charger son propre shader sous Substance Designer, nous avons donc pensé à coder un shader personnalisé capable de faire un rendu du matériau sous la surface de l'eau.

IV. Shading

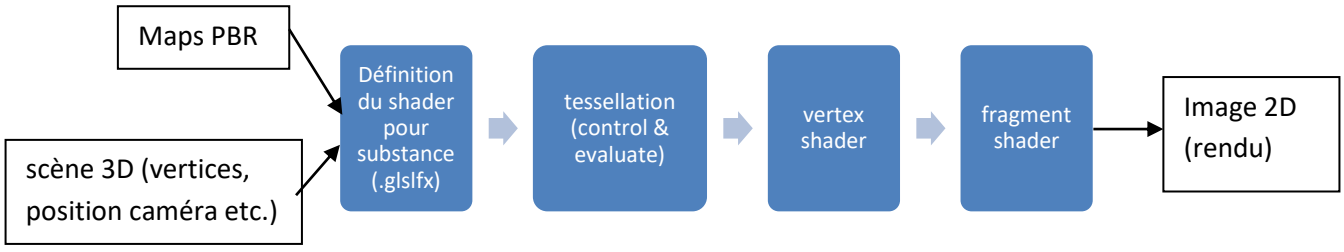
a. Principe

Etant sous OpenGL, les shaders sont codés en GLSL, langage dont je n'avais pas entendu parler auparavant.

Dans les rendus présentés précédemment, le relief est calculé par la tessellation. Cette technique consiste à déplacer les vertices d'un mesh 3D suivant la heightMap fournie, c'est donc une déformation **géométrique** du mesh. La parallaxe, elle, se contente de déplacer les texels suivant la position de la caméra, elle « creuse » plus ou moins le matériau selon sa heightMap pour donner un effet de relief.

L'idée est donc de combiner parallaxe et tessellation pour pouvoir représenter la géométrie du matériau (sec, humide et mouillé + surface de l'eau plane) ainsi que le relief du matériau présent sous la surface de l'eau. Par ailleurs, on veut également conserver les informations de roughness et de normale du matériau immergé.

Voici le processus de calcul d'un rendu fait par le shader :



Etant donné que l'on souhaite avoir plusieurs informations pour un même texel (surface de l'eau et matériau immergé), il faut ajouter en entrée du shader une roughnessMap, une normalMap et une heightMap. Cette dernière ne sera pas destinée à la tessellation mais à la parallaxe.

Figure 24 : heightMap destinée à la tessellation (la zone immergée a une hauteur uniforme)

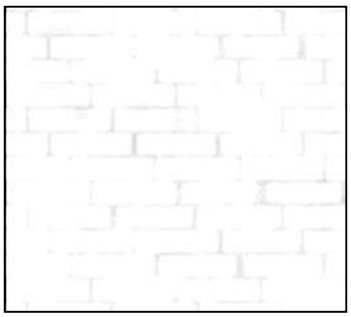


Figure 25 : heightMap destinée à la parallaxe (hauteur du matériau sous la zone immergée)

Enfin, nous avons ajouté deux maps supplémentaires qui définissent le « specularLevel » du matériau et de la couche d'eau. Cette map représente le coefficient F0, soit trivialement l'intensité des réflexions à la normale.

$$F(0°) = (n-1)^2/(n+1)^2$$

n est l'indice du matériau. Pour l'eau, on a donc un coefficient égal à $(1.33 - 1)^2/(1.33 + 1)^2 = 2\%$.

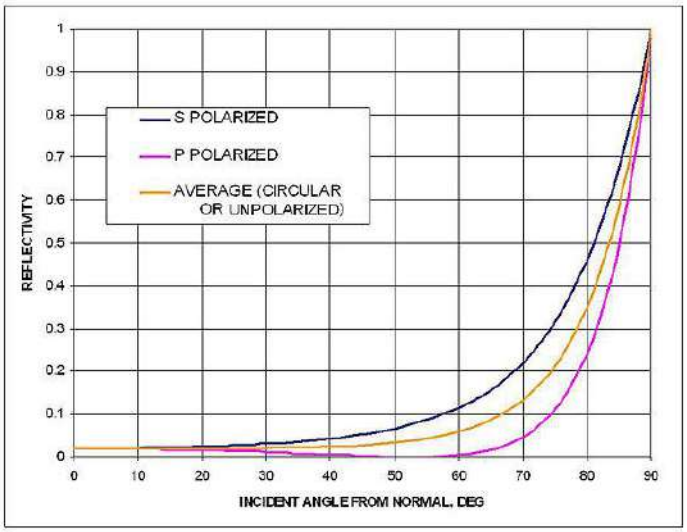


Figure 26 : Réflectivité de l'eau suivant l'angle d'incidence. Ici, F(0°) vaut 2% (0° correspond à la normale)

Vu que l'on souhaite ajouter de la parallaxe au shader de tessellation, nous allons uniquement toucher au fragment shader. C'est un programme écrit en GLSL qui calcule en parallèle la couleur finale de **chaque** pixel du rendu. A la fin du fragment shader sont calculées les différentes contributions lumineuses (de l'environnement, des lumières directionnelles etc.), et la couleur finale du pixel est la somme de ces contributions. Nous nous concentrons ici juste sur la contribution lumineuse de l'environnement. Celle-ci est calculée grâce à la fonction computeIBL (IBL pour image-based lighting)**.**

Mon shader traitera donc la contribution de la couche d'eau et celle du matériau immergé, ces contributions seront appelées respectivement *contribE_water* et *contribE*. Seulement, la quantité de lumière reçue par le matériau immergé a été diminuée par la couche d'eau. Pour cela, on multipliera contribE par un coefficient que l'on présentera plus tard.

b. Sampling

La première étape est de récupérer les valeurs de normal, roughness et height du matériau immergé. Dans le shader, ces maps sont respectivement nommées water_normalMap, water_roughnessMap et water_heightMap.

Ayant la normal map immergée en entrée, je dois récupérer sa couleur dans un vec3 (cela s'appelle du « sampling »). Les coordonnées u et v, définies au début du code, viennent indiquer quel texel je dois récupérer dans la texture.

```
vec3 water_normalTS = get2DSample(water_normalMap, uv,
disableFragment, cDefaultColor.mWater_Normal).xyz;
    if(length(water_normalTS)>0.0001)
    {
     water_normalTS = fixNormalSample(water_normalTS,flipY);
     water_fixedNormalWS = normalize(
      water_normalTS.x*tangentWS +
      water_normalTS.y*binormalWS +
      water_normalTS.z*normalWS );
    }
```

La fonction « get2Dsample » vient sampler la texture en entrée du shader et retourne la couleur aux coordonnées uv dans un vec3. La couleur est ensuite convertie en vecteur (donc remappée), appelé water_normalTS. TS signifie tangent space : le repère géométrique n'est plus celui de la scène 3D (appelé World Space) mais un repère local.

Pour le sampling de la roughness, on utilise aussi la fonction « get2Dsample ».

```
float water_roughness = get2DSample(water_roughnessMap, uv,
disableFragment, cDefaultColor.mWater_Roughness).r;
```

Enfin, passons à la hauteur.

La première étape de la parallaxe est de modifier les coordonnées UV dans le fragment shader courant suivant la position de la caméra pour donner l'impression de relief. Pour ça, je réutilise la fonction *vec2 updateUV(in sampler2D heightmap, in vec3 pointToCameraDirWS, etc.).* La heightMap en entrée sera la heightMap destinée à la parallaxe. Si je veux donner une apparence de réfraction, je dois aussi modifier le vecteur *pointToCameraDirWS,* qui est le vecteur de direction entre la caméra et le vertex traité par le fragment shader.

```
vec3 vec_temp = refract(pointToCameraDirWS,-normalvgWS,(1/1.33));
```

vec_temp sera envoyé dans updateUV, ainsi le décalage UV se fera comme si le vecteur *pointToCameraDirWS* était réfracté.

c. Contributions

La contribution de la couche d'eau est calculée par la fonction « computeIBL ».

```
vec3 contribE_water = computeIBL(
  environmentMap, envRotation, maxLod,
  nbSamples,
  normalWS, water_fixedNormalWS, tangentWS, binormalWS,
  pointToCameraDirWS, false,
  water_diffColor, water_specColor, water_roughness,
  AmbiIntensity * ao);
```

Ensuite, il faut calculer le fameux coefficient qui viendra atténuer la lumière transmise sous la couche d'eau.

```
float ndv_temp = dot(pointToCameraDirWS, water_fixedNormalWS);

vec3 t_01 = vec3(1.0) - fresnel(ndv_temp < 0.0 ? -ndv_temp : ndv_temp, vec3(water_dielectricSpec))
* smoothstep(0.0, 0.001, water_dielectricSpec); //-ndv removes black artifacts at silhouettes.
```

Le coefficient s'appelle « t_01 », pour « lumière transmise entre la couche 0 (air) et 1 (eau) ». Le deuxième terme de la soustraction vaudra 1 aux angles rasants et quasiment 0 à la normale. Cela signifie qu'aux angles rasants, la couche d'eau se comportera comme un miroir : on observera quasiment que les reflets spéculaires, pas le matériau immergé. A la normale, on pourra observer le matériau immergé et les reflets seront très faibles.

Voici la contribution du matériau immergé :

```
vec3 contribE = computeIBL(
  environmentMap, envRotation, maxLod,
  nbSamples,
  normalWS, fixedNormalWS, tangentWS, binormalWS,
  pointToCameraDirWS, true,
  diffColor, specColor, roughness,
  AmbiIntensity * ao) * (t_01);
```

Nous avons ajouté un booléen en paramètre de la fonction computeIBL. Il sert à définir si la lumière a été réfractée ou non, c'est pour cela qu'il vaut « false » pour la première contribution et « true » pour la deuxième.

La couleur finale « ocolor0 » est calculée comme ceci :

```
    vec3 finalColor = contrib0 + contrib1 + contribE +
contribE_water + emissiveContrib;
    float opacity = get2DSample(opacityMap, uv, disableFragment,
cDefaultColor.mOpacity).r;

    ocolor0 = vec4(finalColor, opacity);
```

d. Conclusion sur le shader

Le shader vient résoudre le problème principal de l'ajout d'une couche d'eau au matériau : l'écrasement des informations. Grâce à lui, on peut observer une pièce de monnaie briller au fond de l'eau, car le calcul des contributions se base sur la rugosité de l'eau et du matériau (ici la pièce de monnaie). De même pour la normal map, on peut voir les ombrages et les réflexions propres au matériau immergé. Enfin, le shader représente le relief du matériau immergé, tout en donnant une impression de réfraction.

La fonction main du fragment shader est disponible en **annexe 4**.

V. Améliorations futures

Une amélioration possible est l'ajout d'une map de porosité qui définirait l'absorption du matériau. Ainsi, la zone humide (moist) serait cohérente et calculée automatiquement, de même pour la couleur du matériau mouillé qui est directement corrélée avec l'absorption.

Je n'ai pas eu l'opportunité d'explorer plus en profondeur les phénomènes d'absorption et de diffusion de l'eau. Il serait intéressant d'ajouter des paramètres précis pour les contrôler afin de changer l'apparence du liquide (lait, boue, thé, sang etc.).

Par ailleurs, nous avons observé que des reflets caustiques se formaient sur les bords des gouttes d'eau. Cela rend la goutte plus facile à voir lorsqu'elle est sur un matériau totalement inabsorbant (exemple : plastique ou métal). Nous n'avons pas pris en compte cet effet.

Enfin, la prochaine étape est de publier mon travail sur Substance Share (un site d'échange de Substance, de nœuds, de matériaux etc.). Je souhaite publier mon nœud ainsi que le shader associé, car je pense que cela peut intéresser certains artistes. Pascal a parlé de mon travail à différents développeurs de chez Allegorithmic qui semblaient intéressés, je souhaite donc peaufiner mon travail afin de sortir une version définitive.

V. Conclusion du stage

J'ai eu la chance d'avoir un sujet qui m'a permis d'explorer de nombreux domaines, au sein des quels j'ai développé un grand nombre de compétences et de connaissances. Voici un résumé :

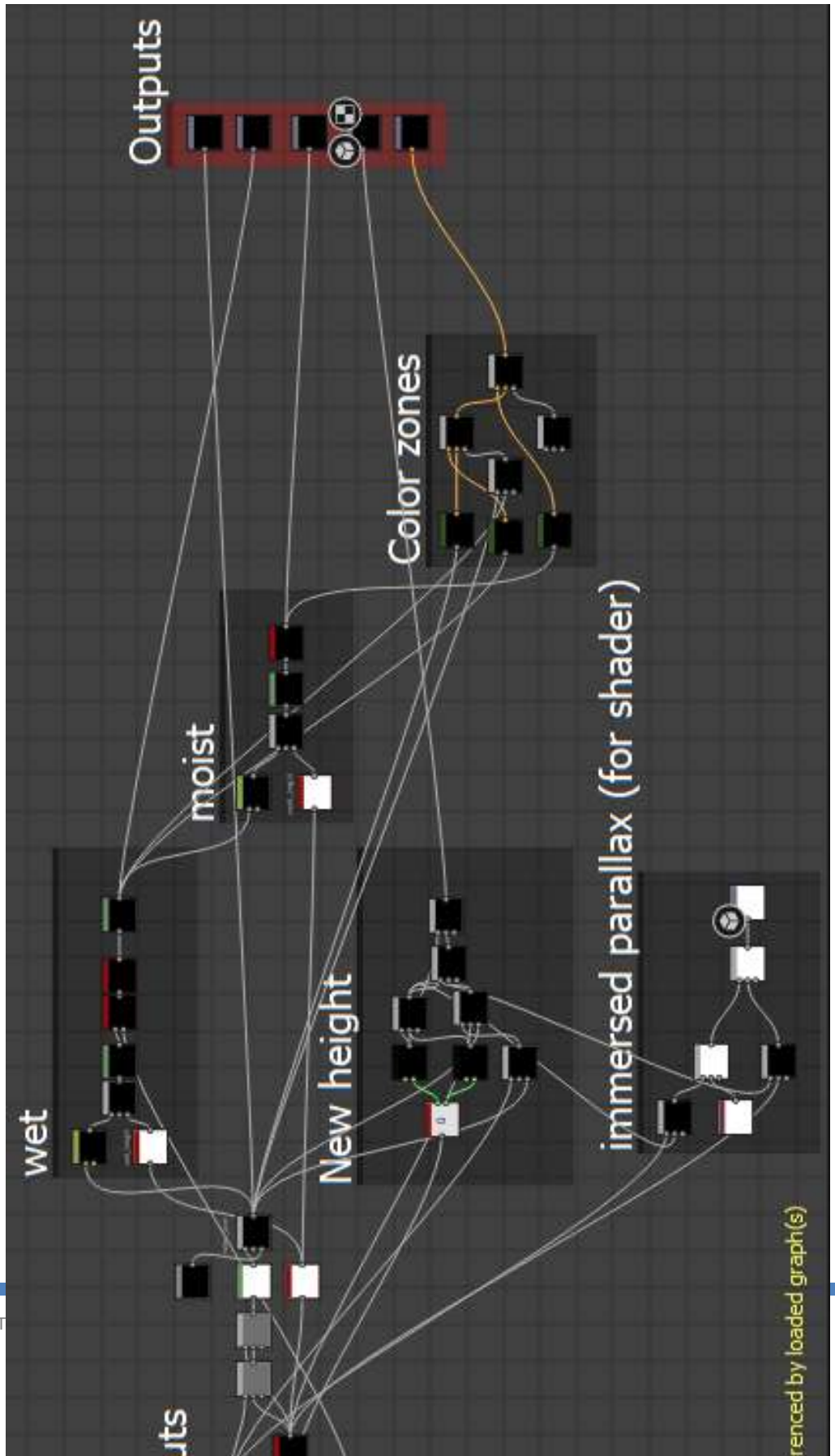
- Maîtrise du logiciel Substance Designer.
- Création d'un nœud « water level » amélioré prenant en compte beaucoup plus de phénomènes physiques, améliorant ainsi le réalisme du rendu.
- Exploration de Substance Painter (logiciel complémentaire à Substance Designer qui permet d'appliquer des matériaux sur des meshes 3D). Tentative de création des zones par la peinture physique de Substance Painter, finalement abandonné.
- Apprentissage du MDL
- Apprentissage du GLSL : développement de mon propre shader basé sur le shader de tessellation.
- Importation de mes matériaux mouillés sur Unity et Unreal Engine (fonctionne)
- Scripting sous Unreal Engine pour augmenter progressivement le niveau de l'eau du matériau.
- Découverte de l'optique : j'ai appris quelles équations et modèles définissent le comportement de la lumière et j'ai pu réinvestir ces connaissances dans le développement de mon shader.

Ma personnalité fait que j'ai eu parfois du mal à rester concentré sur une idée et la poursuivre. J'avais tendance à partir sur différentes idées à la fois, mais j'ai conscience que cela m'a aussi permis d'explorer différentes possibilités dont certaines m'ont permis d'avancer sur mon sujet. J'ai appris à mieux organiser et structurer mes idées. Plutôt que de partir directement dans la pratique comme j'ai été tenté de le faire, j'ai formulé et détaillé le problème le plus possible ce qui m'a donné une ligne directrice que j'ai suivi tout au long du stage. Cela m'a fait réaliser l'importance de procéder comme cela lorsqu'une problématique se présente. De plus, j'ai appris à mathématiser un problème, à voir les équations qui se cachent derrière un phénomène physique ou derrière un obstacle que j'ai du mal à contourner.

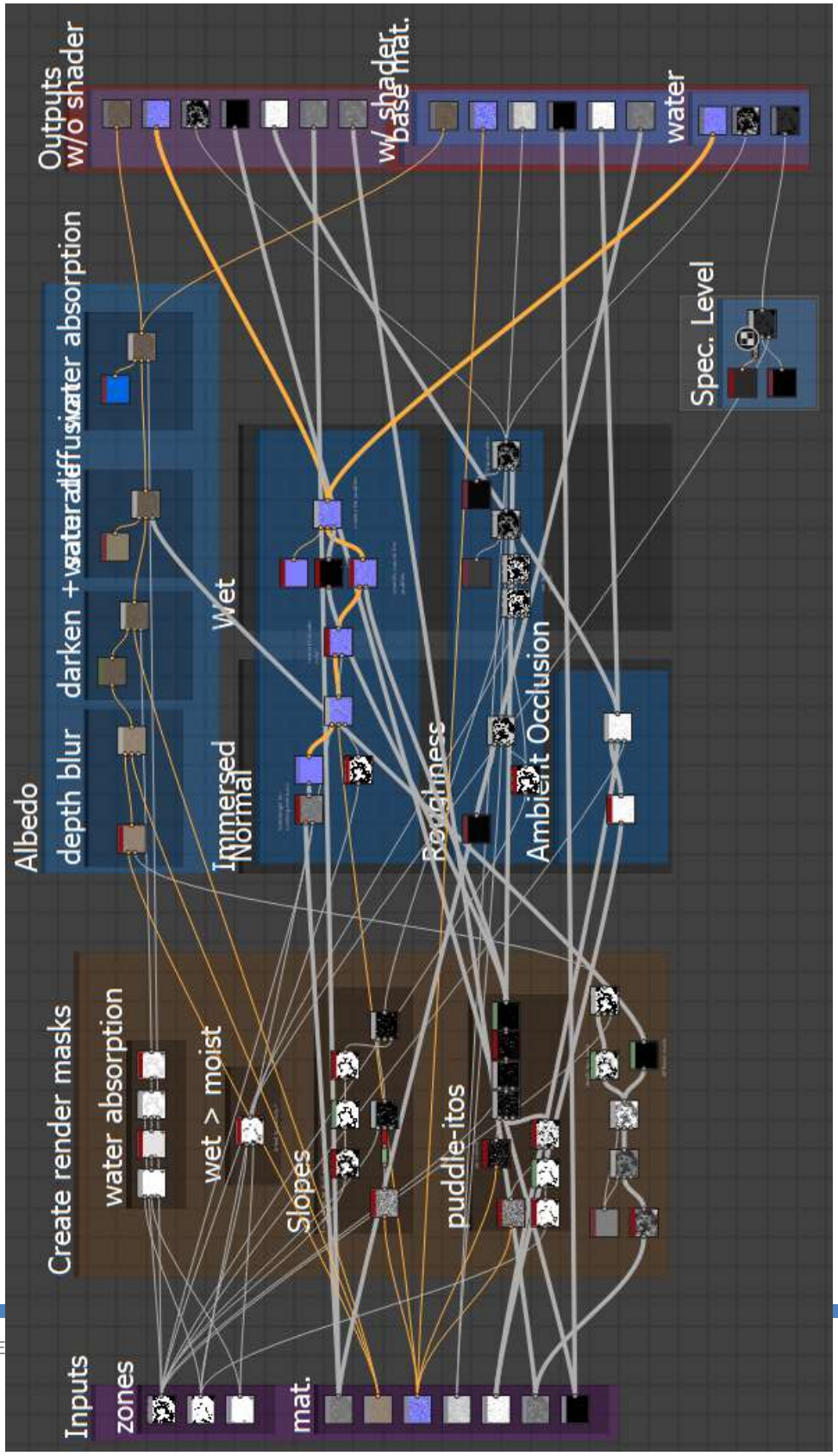
Le monde de la recherche me paraît extrêmement motivant et nutritif, les sujets abordés au sein de l'équipe MANAO sont très variés et les réunions hebdomadaires appelées « GT » (pour groupe de travail) permettent de se tenir informé de l'avancement des chercheurs de l'équipe dans leurs travaux. C'est un monde actif dans lequel on progresse constamment, et c'est ce que je recherche dans mon orientation professionnelle.

En conclusion, ce stage m'a nettement confirmé mon choix de faire le master « image & son » à la suite de ma licence informatique. J'ai trouvé le sujet passionnant, car il permet de faire directement le lien entre la physique et l'informatique. Le monde de l'informatique graphique est fascinant car il est lié à nos perceptions et à notre sensibilité visuelle.

Annexe 1 : le nœud « create masks »



Annexe 2 : le nœud « texturing »



Annexe 3 : quelques rendus (1/2)

Quelques rendus (2/2)

Annexe 4 : Fonction main du fragment shader

```
void main()
{
	vec3 normalWS = iFS_Normal;
	vec3 tangentWS = iFS_Tangent;
	vec3 binormalWS = perFragBinormal ?
		fixBinormal(normalWS,tangentWS,iFS_Binormal) : iFS_Binormal;

	vec3 cameraPosWS = viewInverseMatrix[3].xyz;

	vec3 pointToLight0DirWS = Lamp0Pos - iFS_PointWS;
	float pointToLight0Length = length(pointToLight0DirWS);
	pointToLight0DirWS *= 1.0 / pointToLight0Length;
	vec3 pointToLight1DirWS = Lamp1Pos - iFS_PointWS;
	float pointToLight1Length = length(Lamp1Pos - iFS_PointWS);
	pointToLight1DirWS *= 1.0 / pointToLight1Length;

	vec3 pointToCameraDirWS = normalize(cameraPosWS - iFS_PointWS);

	vec3 water_baseColor = vec3(0,0,0); //The water is black because it doesn't absorb any light
```

```
	// ------------------------------------------
	// Parallax

	vec2 uv;


	vec2 uvScale = vec2(1.0, 1.0);
	if (uvwScaleEnabled)
		uvScale = uvwScale.xy;


	vec3 vec_temp = refract(pointToCameraDirWS,-normalWS,(1/1.33));
	uv = updateUV(
		water_heightMap,
		vec_temp,
		normalWS, tangentWS, binormalWS,
		heightMapScale,
		iFS_UV,
		uvScale,
		tiling);


	uv = uv / tiling / uvScale;
	bool disableFragment = hasToDisableFragment(uv);
	uv = uv * tiling * uvScale;
	uv = getUDIMTileUV(uv);

	// ------------------------------------------
	// Add Normal from normalMap
	vec3 fixedNormalWS = normalWS;  // HACK for empty normal textures
	vec3 normalTS = get2DSample(normalMap, uv, disableFragment, cDefaultColor.mNormal).xyz;
	if(length(normalTS)>0.0001)
	{
		normalTS = fixNormalSample(normalTS,flipY);
		fixedNormalWS = normalize(
			normalTS.x*tangentWS +
			normalTS.y*binormalWS +
			normalTS.z*normalWS );
	}

	vec3 water_fixedNormalWS = normalWS;
	vec3 water_normalTS = get2DSample(water_normalMap, uv, disableFragment,
cDefaultColor.mWater_Normal).xyz;
	if(length(water_normalTS)>0.0001)
	{
		water_normalTS = fixNormalSample(water_normalTS,flipY);
		water_fixedNormalWS = normalize(
			water_normalTS.x*tangentWS +
			water_normalTS.y*binormalWS +
			water_normalTS.z*normalWS );
	}

	// ------------------------------------------
```

```
	// Compute material model (diffuse, specular & roughness)
	float dielectricSpec = get2DSample(specularLevel, uv, disableFragment, cDefaultColor.mSpecularLevel).r;
	dielectricSpec = 0.08 * dielectricSpec;
	vec3 dielectricColor = vec3(dielectricSpec, dielectricSpec, dielectricSpec);

	float water_dielectricSpec = get2DSample(water_specularLevel, uv, disableFragment,
cDefaultColor.mWater_SpecularLevel).r;
	water_dielectricSpec = 0.08 * water_dielectricSpec;
	vec3 water_dielectricColor = vec3(water_dielectricSpec, water_dielectricSpec, water_dielectricSpec);

	const float minRoughness=1e-4;
	// Convert the base color from sRGB to linear (we should have done this when
	// loading the texture but there is no way to specify which colorspace is
	// used for a given texture in Designer yet)
	vec3 baseColor = get2DSample(baseColorMap, uv, disableFragment, cDefaultColor.mBaseColor).rgb;
	if (sRGBBaseColor)
		baseColor = srgb_to_linear(baseColor);


	float metallic = get2DSample(metallicMap, uv, disableFragment, cDefaultColor.mMetallic).r;
	float roughness = get2DSample(roughnessMap, uv, disableFragment, cDefaultColor.mRoughness).r;
	roughness = max(minRoughness, fit_roughness(roughness));

	vec3 diffColor = baseColor * (1.0 - metallic);
	vec3 specColor = mix(dielectricColor, baseColor, metallic);

	vec3 water_diffColor = water_baseColor;
	vec3 water_specColor = mix(water_dielectricColor, water_baseColor, vec3(0,0,0));

	// ----------------------------------------
	// Compute point lights contributions
		[…]

	// ----------------------------------------
	// Image based lighting contribution

	float ao = get2DSample(aoMap, uv, disableFragment, cDefaultColor.mAO).r;

	//[CUSTOM] contribution couche d'eau
	float water_roughness = get2DSample(water_roughnessMap, uv, disableFragment,
cDefaultColor.mWater_Roughness).r;

	vec3 contribE_water = computeIBL(
		environmentMap, envRotation, maxLod,
		nbSamples,
		normalWS, water_fixedNormalWS, tangentWS, binormalWS,
		pointToCameraDirWS, false,
		water_diffColor, water_specColor, water_roughness,
		AmbiIntensity * ao);

	float ndv_temp = dot(pointToCameraDirWS, water_fixedNormalWS);
	vec3 t_01 = vec3(1.0) - fresnel(ndv_temp < 0.0 ? -ndv_temp : ndv_temp, vec3(water_dielectricSpec)) *
smoothstep(0.0, 0.001, water_dielectricSpec); //-ndv removes black artifacts at silhouettes.
```

```
        vec3 contribE = computeIBL(
                environmentMap, envRotation, maxLod,
                nbSamples,
                normalWS, fixedNormalWS, tangentWS, binormalWS,
                pointToCameraDirWS, true,
                diffColor, specColor, roughness,
                AmbiIntensity * ao) * (t_01);

        // ----------------------------------------
        //Emissive
        vec3 emissiveContrib = get2DSample(emissiveMap, uv, disableFragment, cDefaultColor.mEmissive).rgb;
        emissiveContrib = srgb_to_linear(emissiveContrib) * EmissiveIntensity;

        // ----------------------------------------
        vec3 finalColor = contrib0 + contrib1 + contribE + contribE_water + emissiveContrib; // + contribE
        // Final Color
        // Convert the fragment color from linear to sRGB for display (we should
        // make the framebuffer use sRGB instead).
        float opacity = get2DSample(opacityMap, uv, disableFragment, cDefaultColor.mOpacity).r;

        ocolor0 = vec4(finalColor, opacity);
        ocolor0 = outcolor;
}
```