Shader, Texture et Eclairage

Graphics Programming

Eric Cannet

But de ce cours

- Win 32
- Texture
- Shaders
- Éclairage

Win 32

#include<windows.h>

- Win32 est une bibliothèque windows
- Venu avec Windows 95
- Elle permet d'interagir avec le système et de faire des applications graphique
- Un peu deprecated pour faire une application windows avec des fenêtres partout
- Le man c'est la MSDN

int main(int argc, char** argv)

- Le point d'entré du programme n'est plus main mais:

```
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow );
```

Quelques types

- Quelques types de données pour pas être trop perdu:
  - BYTE, WORD, DWORD
  - LPSTR, LPCSTR
  - HANDLE ou H*, HINSTANCE, HWND
  - LPVOID
  - UINT
  - HRESULT

Quelques conventions

- Quelques conventions Windows et DirectX:
  - Nom de type commencent par I : interface
  - Nom de type commencent par LP : pointeur
    - Donc LPTOTO * est un pointeur de pointeur vers TOTO
  - Variable commençant par p :pointeur
  - Variable commençant par pp :pointeur de pointeur

```
LPTOTO* ppTOTO = NULL;
Afunction(ppTOTO );
```

Le pointeur a été allouer et remplie

Windows montre toi

- Pour créer une fenêtre il y toutes ces étapes :
  - Créer un type de fenêtre (identifié par un jolie nom)
  - Créer la fenêtre en spécifiant entre autre le nom de son type de fenêtre
  - Lui dire de ce montrer

  - Mettre la jolie boucle infini qui traite les évènements

Ho un message ! J'ai des amis ?

- Windows fonctionne avec un système de message.
- Dès qu'il se passe quelque chose, Windows vous en informe via un message.

- Un message se traduit par l'appel de la fonction de traitement de message que vous lui avez indiqué lors de la création de la fenêtre

Zut orange info

```
MSG oMsg;
PeekMessage( &oMsg, NULL, 0, 0, PM_NOREMOVE );
while ( oMsg.message != WM_QUIT )
{
   if (PeekMessage( &oMsg, NULL, 0, 0, PM_REMOVE ))
   {
         TranslateMessage( &oMsg );
         DispatchMessage( &oMsg );
   }
   else
   {
         RenderOneFrame();
   }
}
```

Répondeur automatique

```
LRESULT WINAPI MsgProc (HWND hwnd, UINT
   msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
         case WM_DESTROY:
                  PostQuitMessage(0);
                  break;
   }
   return
   DefWindowProc(hwnd,msg,wParam,lParam);
}
```

Les Textures

Entrez dans la 2e dimension

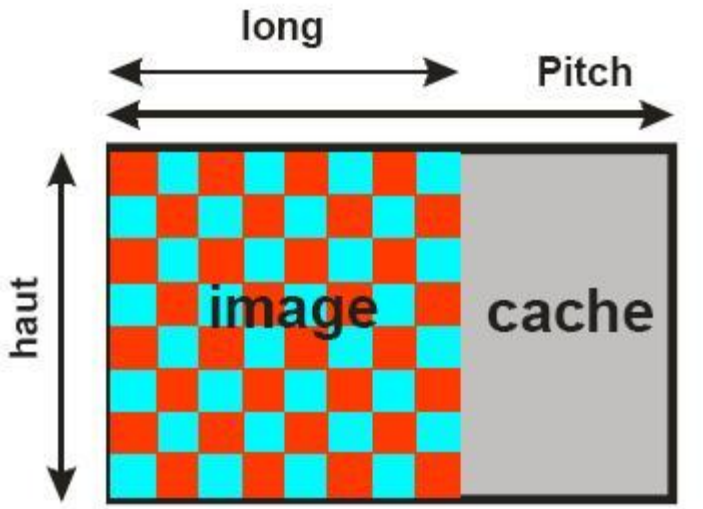
- Il existe différents types de texture :
  - Texture 1D
  - Texture 2D
  - Cube Map
  - Texture 3D
  - TextureArray (à partir de DirectX 10)

R2D2

- Il existe une multitude de format (liste non exhaustive):
  - Int :
    - A8R8G8B8
    - X8R8G8B8
  - Float :
    - R32F
    - A32B32G32R32F
  - Autre:
    - DXT1
    - DXT5
  - Spéciale:
    - D24S8
    - D32

Un pitch dans ta poche

- Les textures sont en taille de puissance de 2 dans la mémoire, il y a donc un pitch :

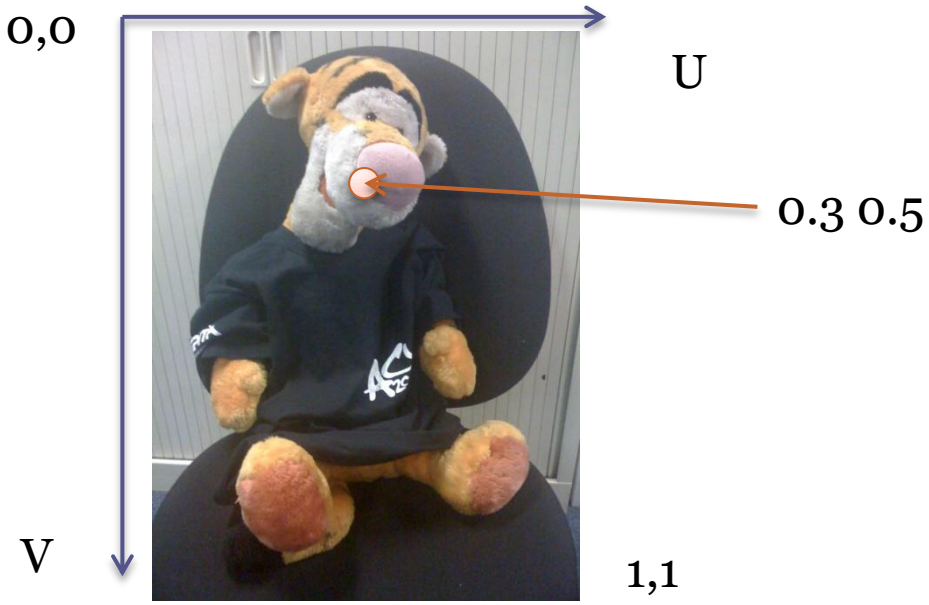


surface associée à une image

Cool mais comment on map

- Les vertex vont contenir des coordonnées de texture appelés UV.

- Ces coordonnées sont comprise entre 0 et 1.

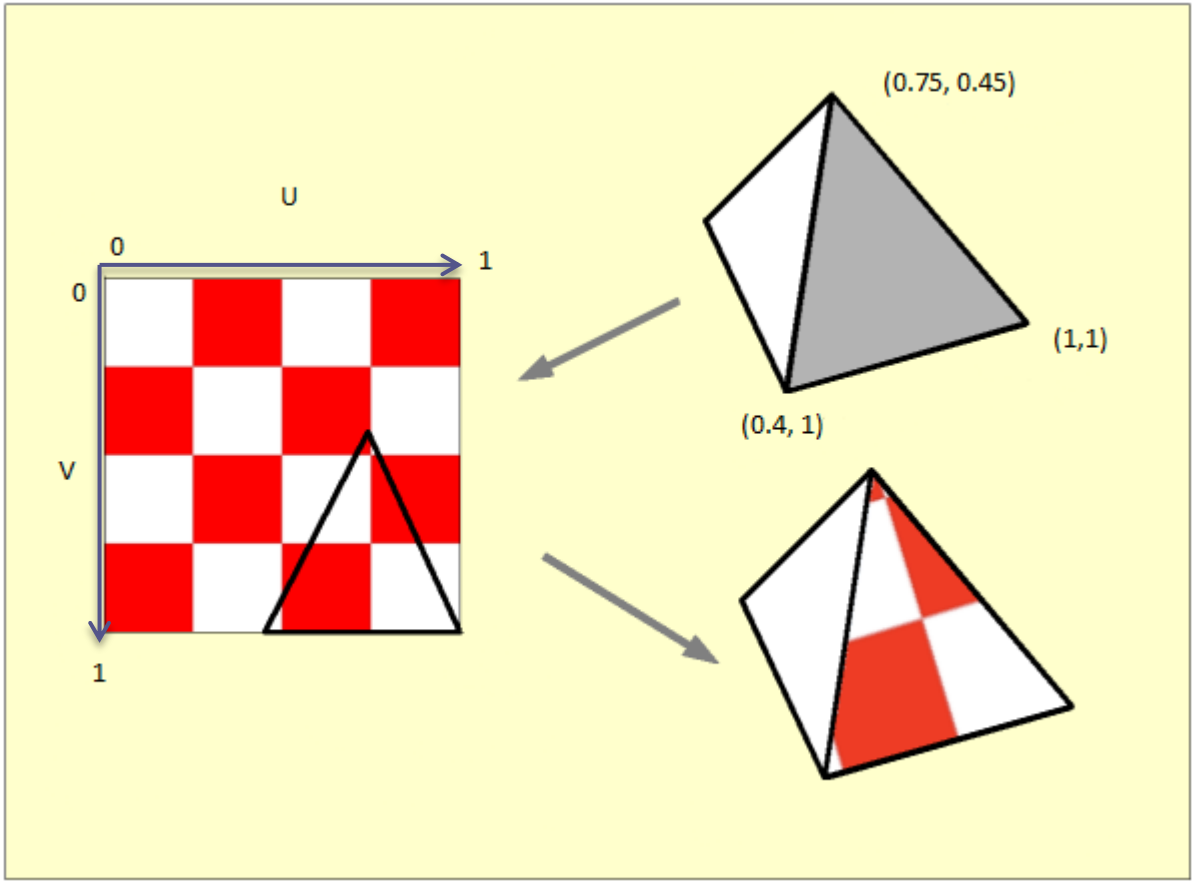


Rasterisation d'UV

- Lors de la Rasterisation ces coordonnées UV vont être interpolées linéairement.

- Au Pixel Shader on aura alors à demander la couleur du pixel de la texture se trouvant aux coordonnées UV

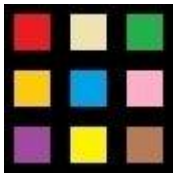
UV mapping en image

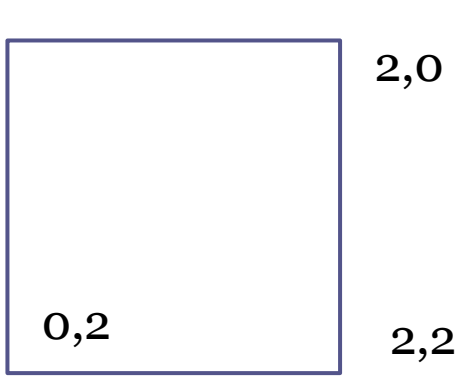


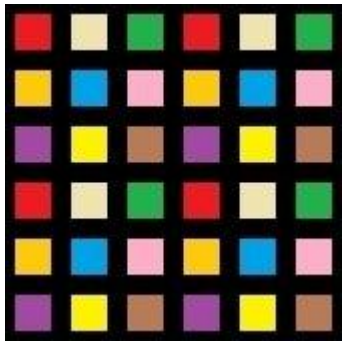
Et au bord on tombe?

- Que ce passe-t-il quand un UV n'est pas compris entre 0 et 1 ?
  - Clamp
  - Wrap
  - Clamp to border

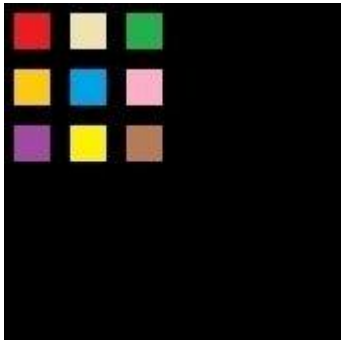
Exemples de chutes



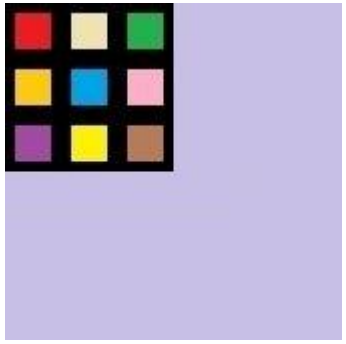




Wrap



Clamp



Clamp to border

Mini moi

- Les Mip Map sont l'ensemble des surfaces qui représente la même texture mais à des tailles différentes.
- A chaque MipMap la taille est divisée par 2

- Permet d'avoir un gain en performance et en qualité

Mince! elle n’est pas en entier

MipMap Moi

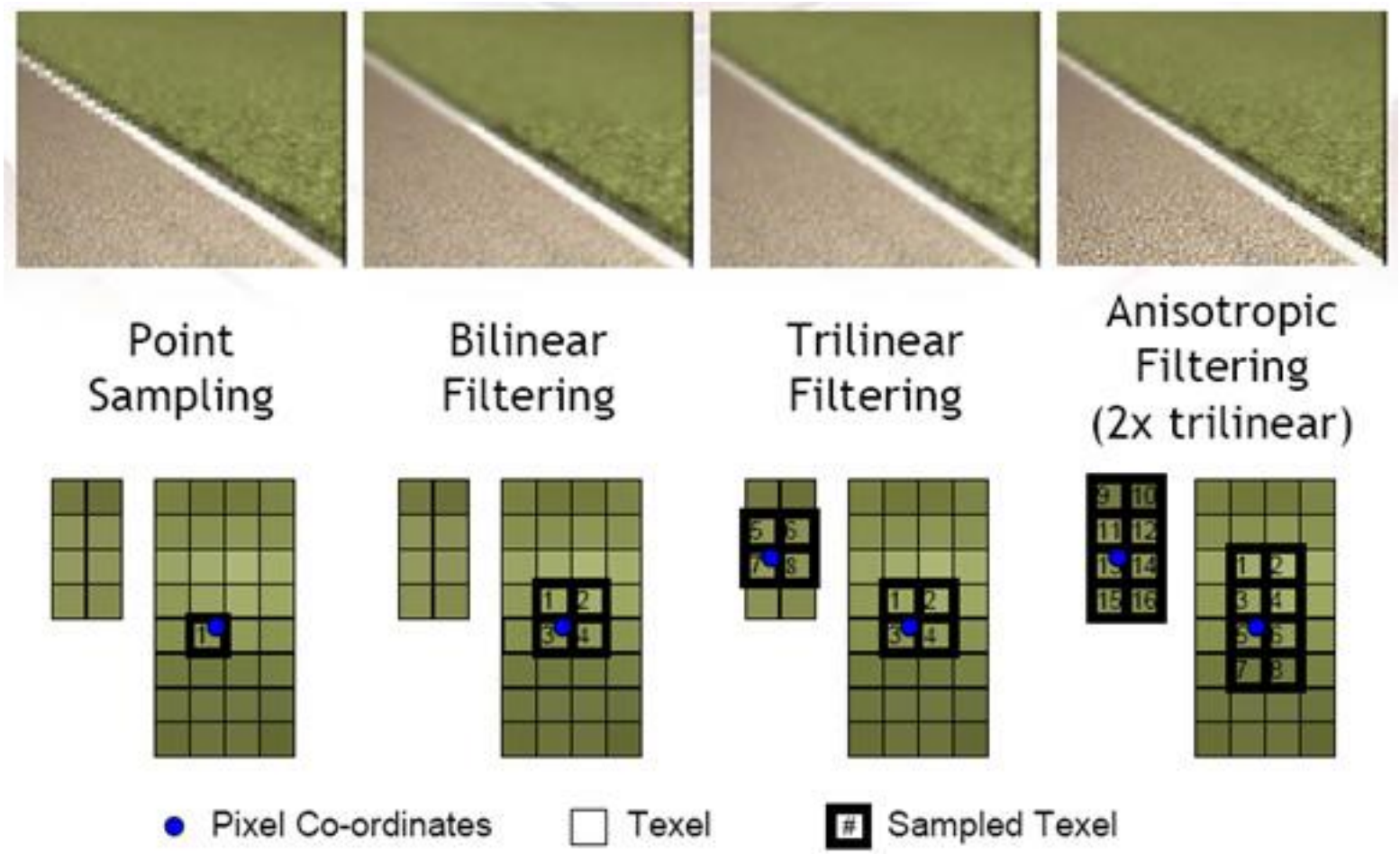
Texture Filtering

- Deux catégories:
  - Magnification: Un pixel écran correspond à moins d'un pixel de la texture
  - Minification: Un pixel écran correspond à plus d'un pixel de la texture
- Différentes filtres (Les plus connues):
  - None (que pour les mipmap)
  - Nearest, Point
  - Linear
  - Bilinear
  - Trilinear
  - Anisotropic

Filtres


Bilinear
Trilinear
4x Anisotropic
16x Anisotropic
Free image hosting by
www.techpowerup.com

Explications des filtres



Les Shaders

HLSL

- Il s'agit d'un petit programme exécuté sur la carte graphique qui fait une étape du pipeline graphique

- Ils peuvent être écrit en:
  - Asm
  - HLSL
  - GLSL
  - Cg
- On va voir du HLSL model 3 (DirectX 9)

Type de données

- *bool* - true or false.
- *int* - 32-bit signed integer.
- *uint* - 32-bit unsigned integer.
- *half* - 16-bit floating point value.
- *float* - 32-bit floating point value.
- *Double* - 64-bit floating point value.

- Vecteur : typeN (Exemple float4)
- Matrice : typeNxN (Exemple float4x4)

Contrôle du flux

- Contrôle du flux:
  - If, else
  - Switch case
  - Do, while, for
- Syntaxe quasi identique au C
- On peut spécifier des optimisations (liste non exhaustive)
  - Unroll
  - Branch, flatten

Les Variables

- *Comme on C :*

  [Storage_Class] [Type_Modifier] Type Name[: Semantic] [= Initial_Value]

Exemples :

```
shared float3 camPosition = float3(1.0f, 0.3f,0.0f);
const float lodDist[3]  = {10.1f, 666.0f, 1664.0f};
float4x4 world;
```

Struct

- Comme en C:

struct *Name*
{
[InterpolationModifier] Type[RxC]
  MemberName;
  …
};

Exemple

```
struct VertexInput
{
   float4 Position          : POSITION;
   float3 Normal            : NORMAL;
   float2 UV                : TEXCOORD0;
   float4 Weight            : BLENDWEIGHT;
   int4 BoneIndices         : BLENDINDICES;
};
```

Fonction

- Presque comme en C

[StorageClass]
Return_Value *Name ([ArgumentList]) [: Semantic]*
{
[StatementBlock]
};

- Les paramètres :

[InputModifier] Type Name [: Semantic] [InterpolationModifier] [= Initializers]

Exemple

```
inline float MyFunction(float3 Position : POSITION,
   Float2 UV : TEXCOORD0 = float2(0.0f,0.0f), out
   VertexOutput output)
{
   output.Position = mul( float4(Position, 1.0f),
   WorldViewProj );
   output.UV.xyz = UV.zyy;

   if (output.UV.x *  2.0f > 20)
         output.UV.w = max(0.45f, dot(output.UV.xyz,
   Position.xxx));

   return 1.0f;
}
```

Texture

- Déclaration d'une texture :

Texture2D Name;

- Déclaration d'un sampler

sampler *Name* =
SamplerType
{
  Texture = <*texture_variable*>;
  [*state_name* = *state_value;*]
  …
};

Exemple

```
texture2D   DiffuseMap;
sampler2D DiffuseMapSampler = sampler_state
{
   Texture    =        <DiffuseMap>;
   MinFilter =        LINEAR;
   MagFilter=        LINEAR;
   MipFilter =        LINEAR;
   AddressU=        WRAP;
   AddressV =        WRAP;
};
```

Exemple d’utilisation

```
Float4 color = tex2D( DiffuseMapSampler,
   input.UV );
```

Intrinsics

- Plein d'intrinsics (liste non exhaustive):
  - Cos, sin, tan, acos, asin, atan
  - Min, max, clamp, saturate
  - Ceil, floor, round
  - Lerp,
  - Log, exp, pow, sqrt
  - Mul, transpose
  - Normalize

  - Etc

Effet

- Un effet peut contenir plusieurs Techniques.
- Une Technique est un rendu particulier. Cela comporte plusieurs Techniques Pass.
- Une Technique Pass est un ensemble Vertex et Pixel Shader

Technique

- Syntaxe pour une Technique:

technique [id] [< annotation(s) >]
{
pass(es)
}

- Syntaxe pour une Technique pass:

pass [id] [< annotation(s) >]
{
state assignment(s)
}

Exemple

```
technique diffuse
{
  pass p0
  {
    VertexShader = compile vs_3_0 DiffuseVS();
    PixelShader  = compile ps_3_0 DiffusePS();
  }
}
```

Vertex Shader

- Les structures de données:

```
shared          matrix WorldViewProj;

struct VertexInput
{
   float3 Position      : POSITION;
   float3 Color         : COLOR0;
};

struct VertexOutput
{
   float4 Position      : POSITION;
   float3 Color         : COLOR0;
};
```

Vertex Shader

- La fonction:

```
VertexOutput DiffuseVS(VertexInput input)
{
  VertexOutput output;

  output.Position = mul( float4(input.Position, 1.0f),
  WorldViewProj );
  output.Color = input.Color;

  return output;
}
```

Pixel Shader

- Le Pixel Shader

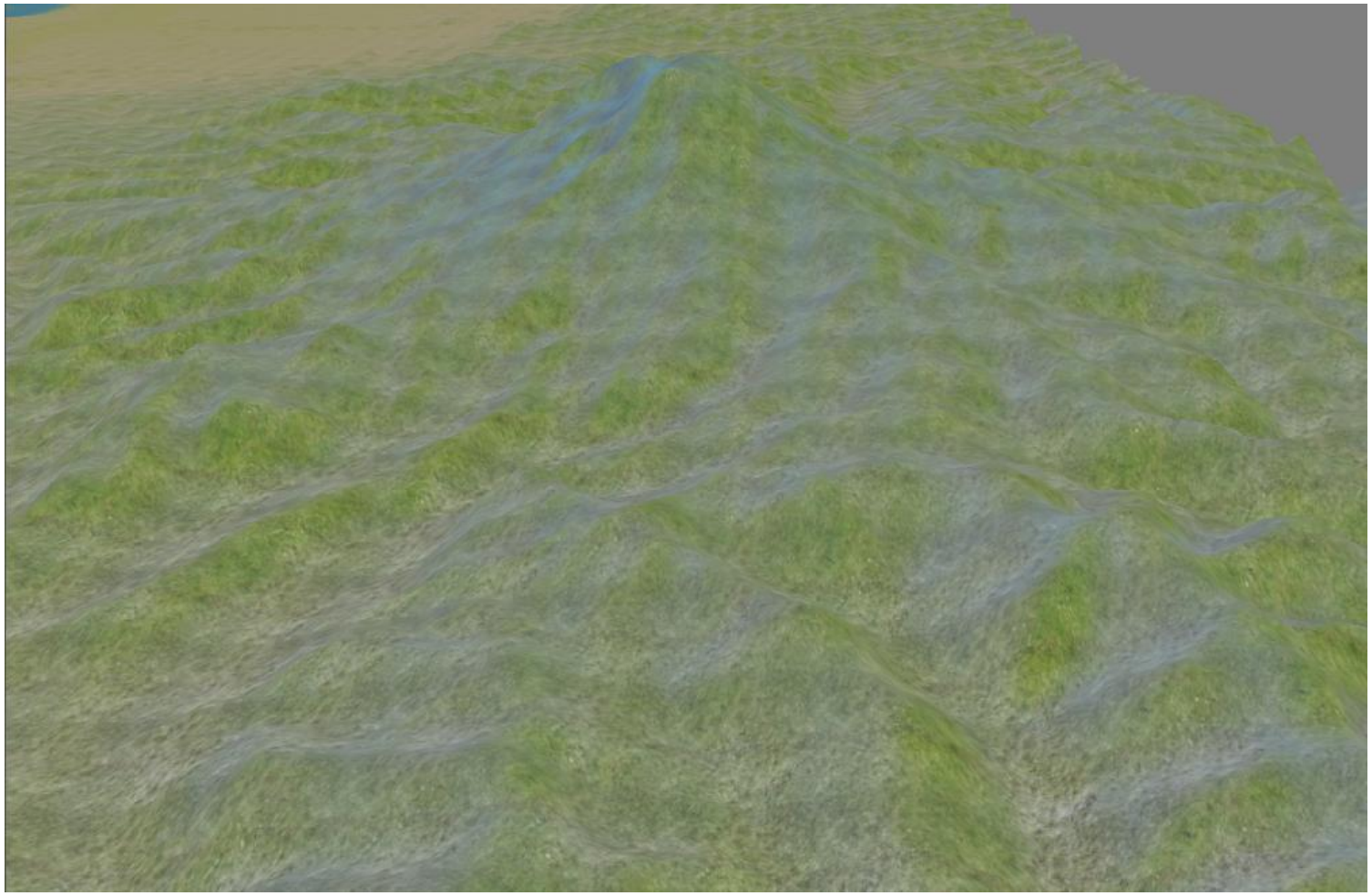
```
float4 DiffusePS( VertexOutput input) : COLOR0
{
  //return float4(1.0f, 1.0f,0.0f, 1.0f);
  return float4(input.Color, 1.0f);
}
```

L’éclairage

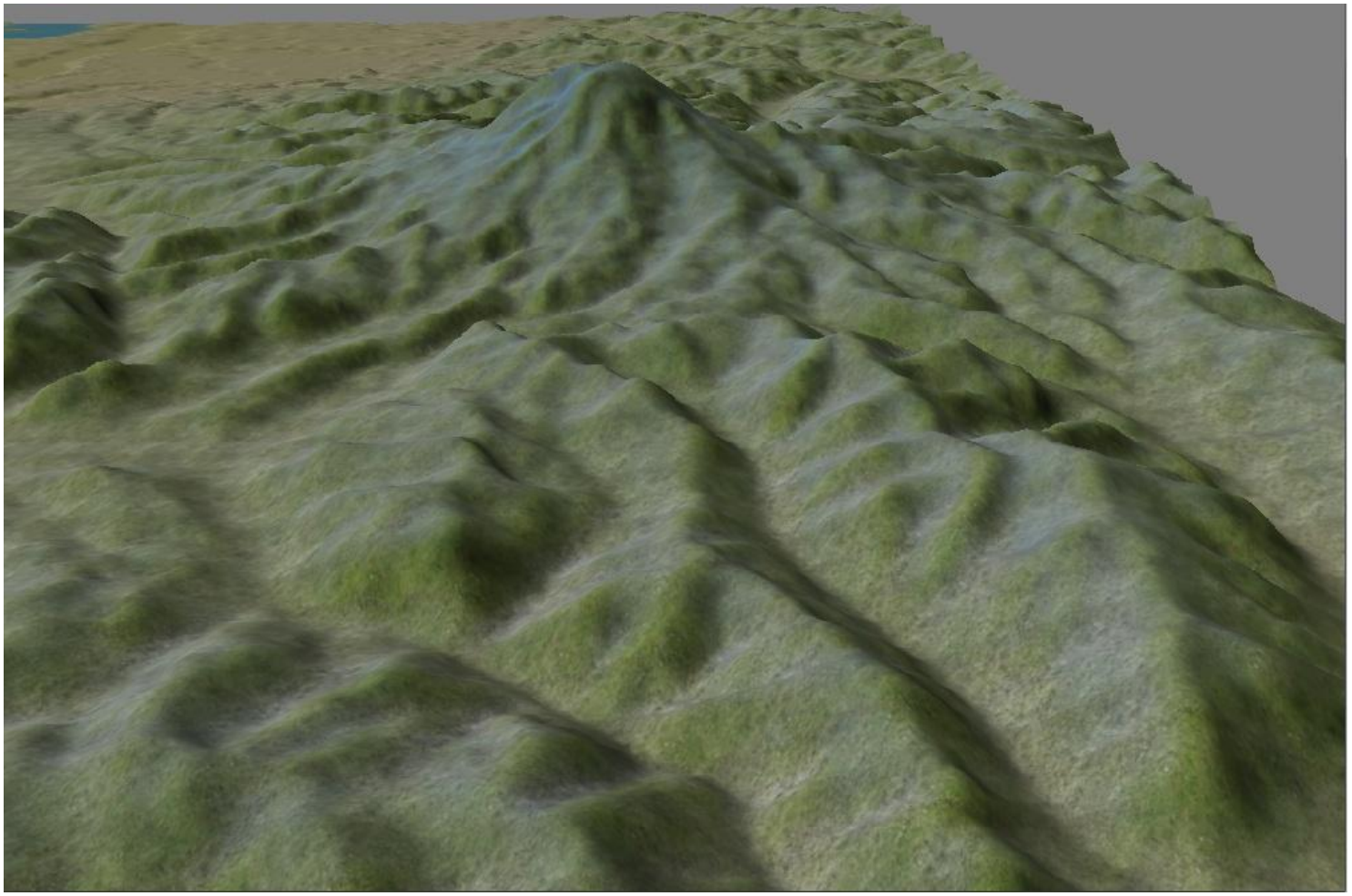
Pourquoi ?

- La lumière est très important :
  - Donne du relief, des détails, de la hauteur
  - Fait plus réaliste
  - Donne une ambiance

Sans

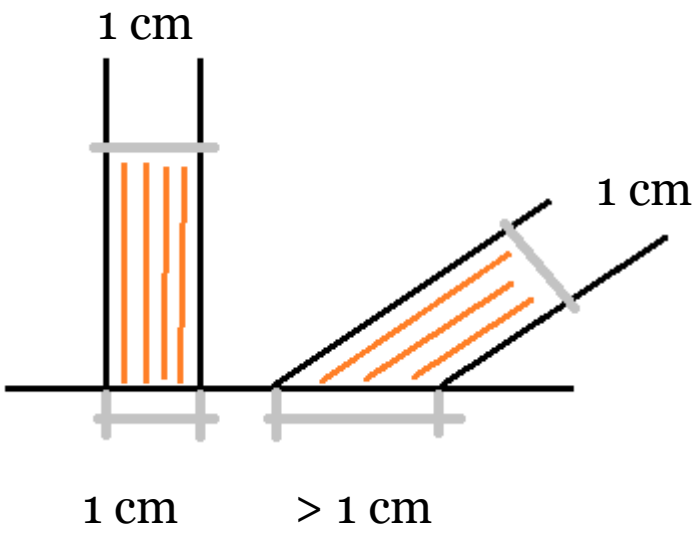


Avec



Lumière diffuse

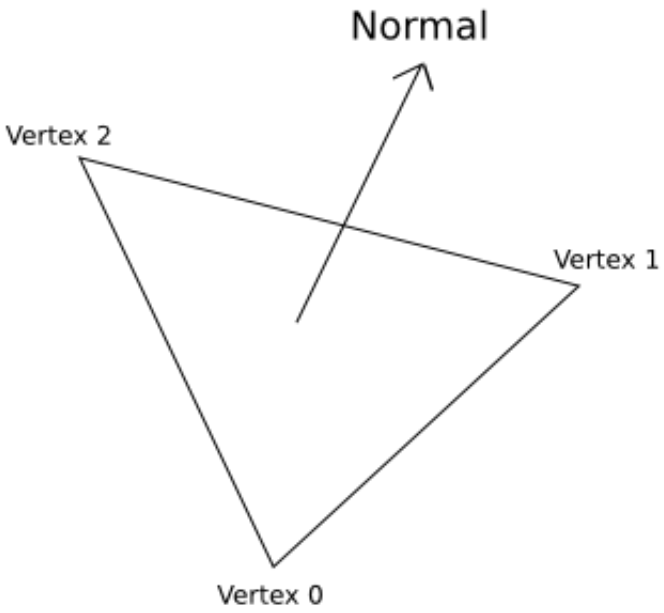
- La lumière se déplace en ligne droite. Plus une surface est perpendiculaire aux rayon lumineux plus elle reçoit de lumière.



- Les calcul d'éclairage sont basés sur les normales

Normal, vous avez dit normal

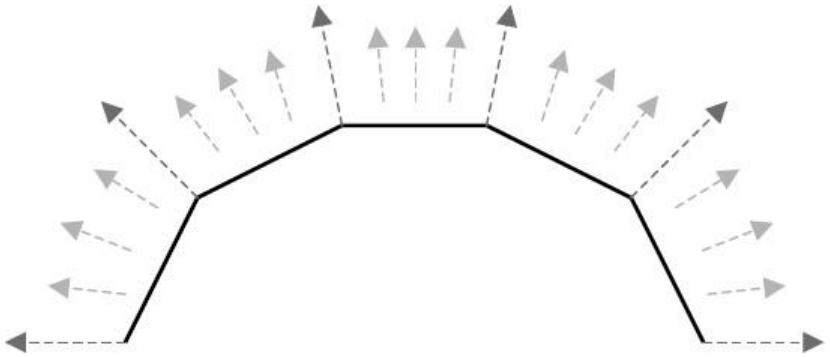
- Chaque sommet contient une normal normalisés



Rasterison une normale

- À la Rasterisation les normals sont interpolés linéairement.

- Ce qui veut dire que cela ne forme pas une courbe



1,2 et 3 zéro

- En général on distingue 3 types de lumière:
  - Spot (une lampe torche)
    - Un point, la position
    - Et deux angles le cône interne et externe
  - Omnidirectionnelle (une ampoule, une torche)
    - Un point, la position
    - Distance d'atténuation
  - Directionnelle (le soleil)
    - Une direction

Et de quatre

- La plus part du temps on a une directionnelle et plusieurs omnidirectionnelle
- D'autre type sont apparue pour n'en cité qu'un : Quad light

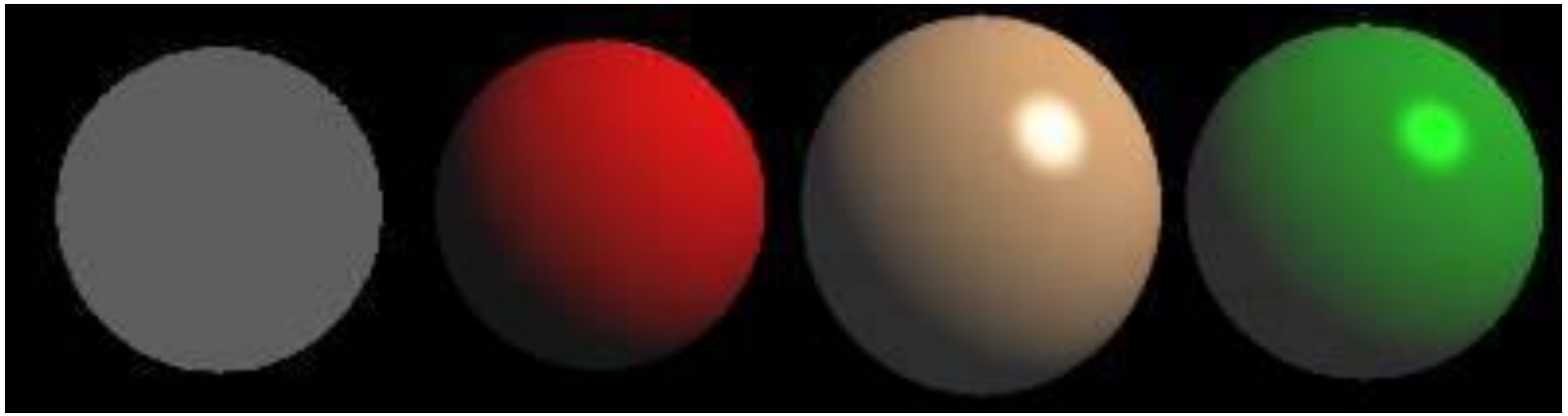
Aziz Lumière !!

- La couleur final est souvent calculée de cette façon :

CouleurFinal =
CouleurDiffuse* EclairageAmbiante+
CouleurDiffuse* EclairageDiffuse+
CouleurSpeculaire* EclairageSpeculaire+
Emmisive

- Mais chacun fait comme il veut.

Aziz une image !!



Diffuse et Specular

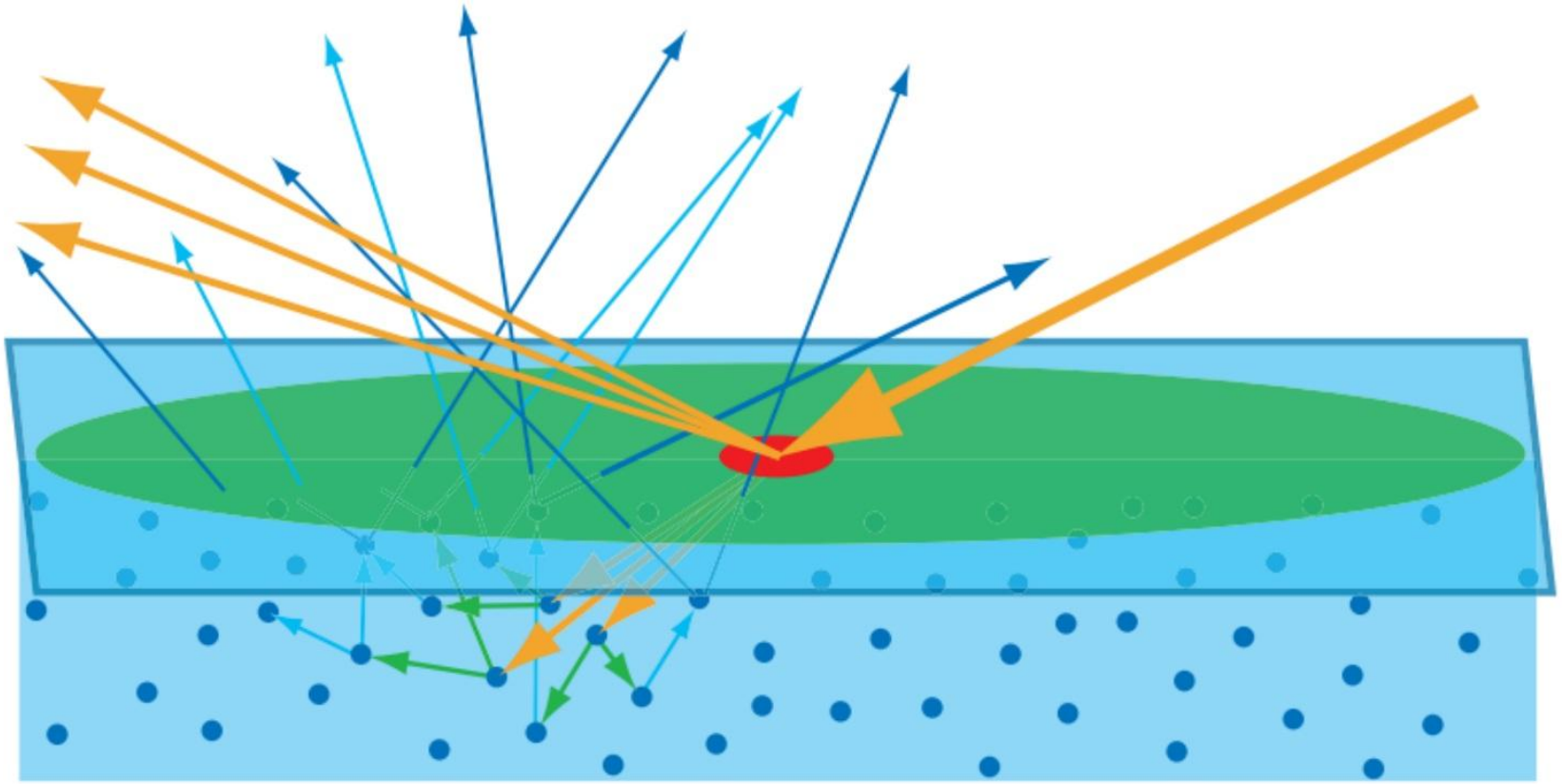


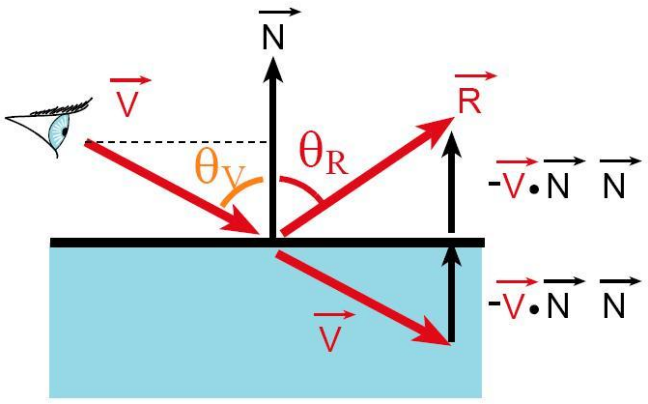
Image from “Real-Time Rendering, 3rd Edition”, A K Peters 2008

Diffuse

- La couleur et l'éclairage de base de l'objet.
- Eclairage diffuse dépend directement de l'orientation de la normal par rapport à celle de la lumière:
  - Produit scalaire de la normal et du vecteur direction de la lumière
- On peut multiplier par l'atténuation ou encore la couleur diffuse de la lumière

Speculaire

- Correspond au reflet.
- Eclairage speculaire: Produit scalaire du vecteur de réflexion de la lumière et du vecteur de direction de la vue
- Vecteur de réflexion : R = -Dl + (2 * N * (Dl).N)
- On peut multiplier par l'atténuation ou encore la couleur spéculaire de la lumière
- Le coefficient est mis à une puissance



Spectaculaire ou speculaire

Forward rendering

- On affiche tous les objets et on calcule tout l'éclairage dessus en même temps, potentiellement en plusieurs passes.

- Complexité : Nb objet * Nb lumière

Forward mais où?

- Ou fait-on le calcul de l'éclairage ?

- Vertex Shader:
  - Moins couteux donc bien pour les objets loin
  - Il faut pas oublié que se sera linéairement interpolé par le Rasteriser
  - Pas de specular
- Pixel Shader:
  - Permet de faire des techniques comme le Bump Mapping
  - Si au vertex alors c'est linéaire

LightMap

- Pour les objets statiques et les lumières statiques l'éclairage ne changera pas. On peut donc la pré calculer.
- On a une texture supplémentaire contenant l'éclairage pré calculé que l'on applique comme une texture classique.

- En revanche il ne faut pas que 2 sommets distinct ai les mêmes coordonnées de texture

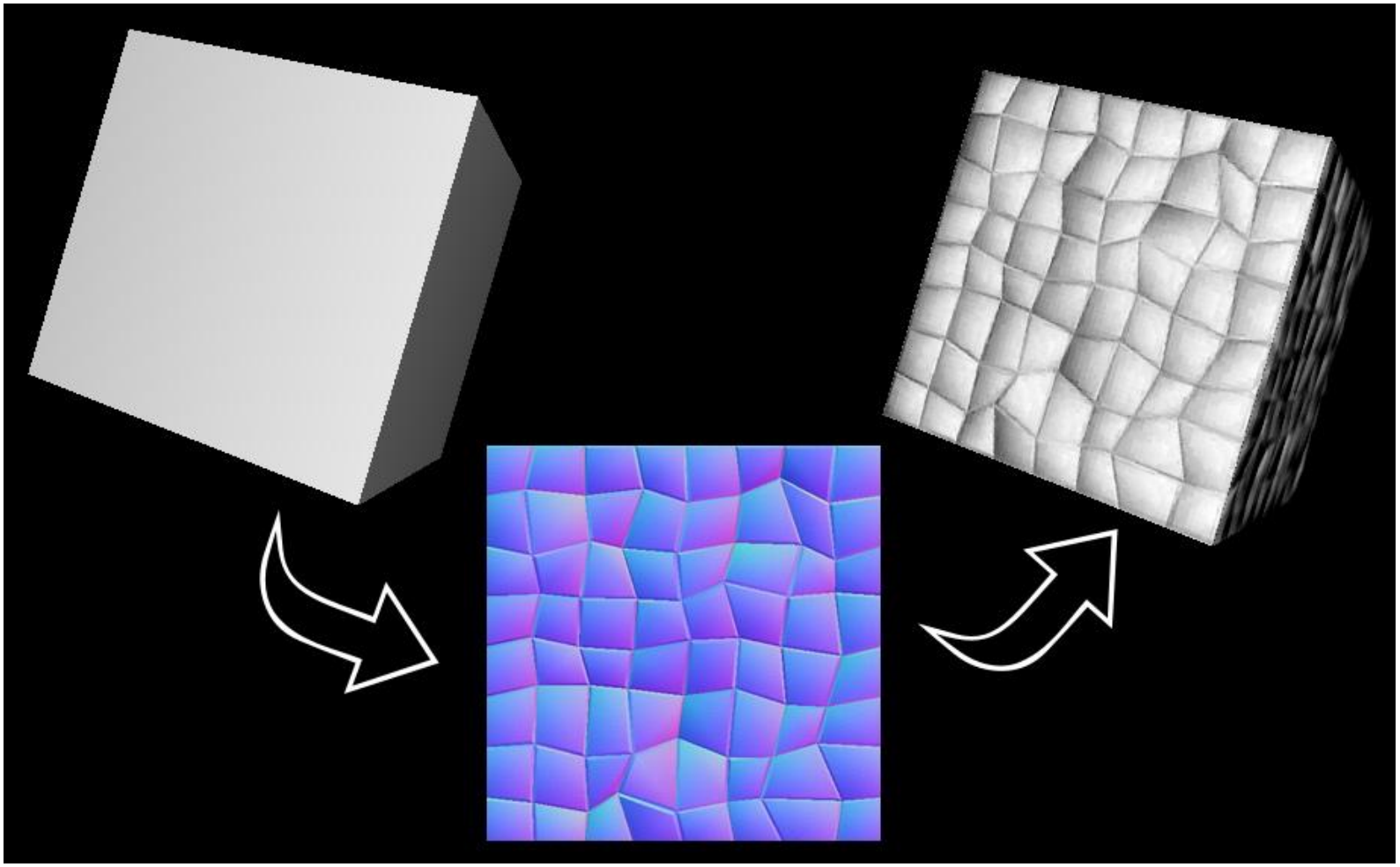
Bump Mapping

- Simuler du relief grâce à l'éclairage.
- On modifie la normal au pixel. Chaque pixel aura une normal. Cette normal viens d'une texture : NormalMap
- Cette normal est représentée dans le repère Tangente, Binormal, Normal

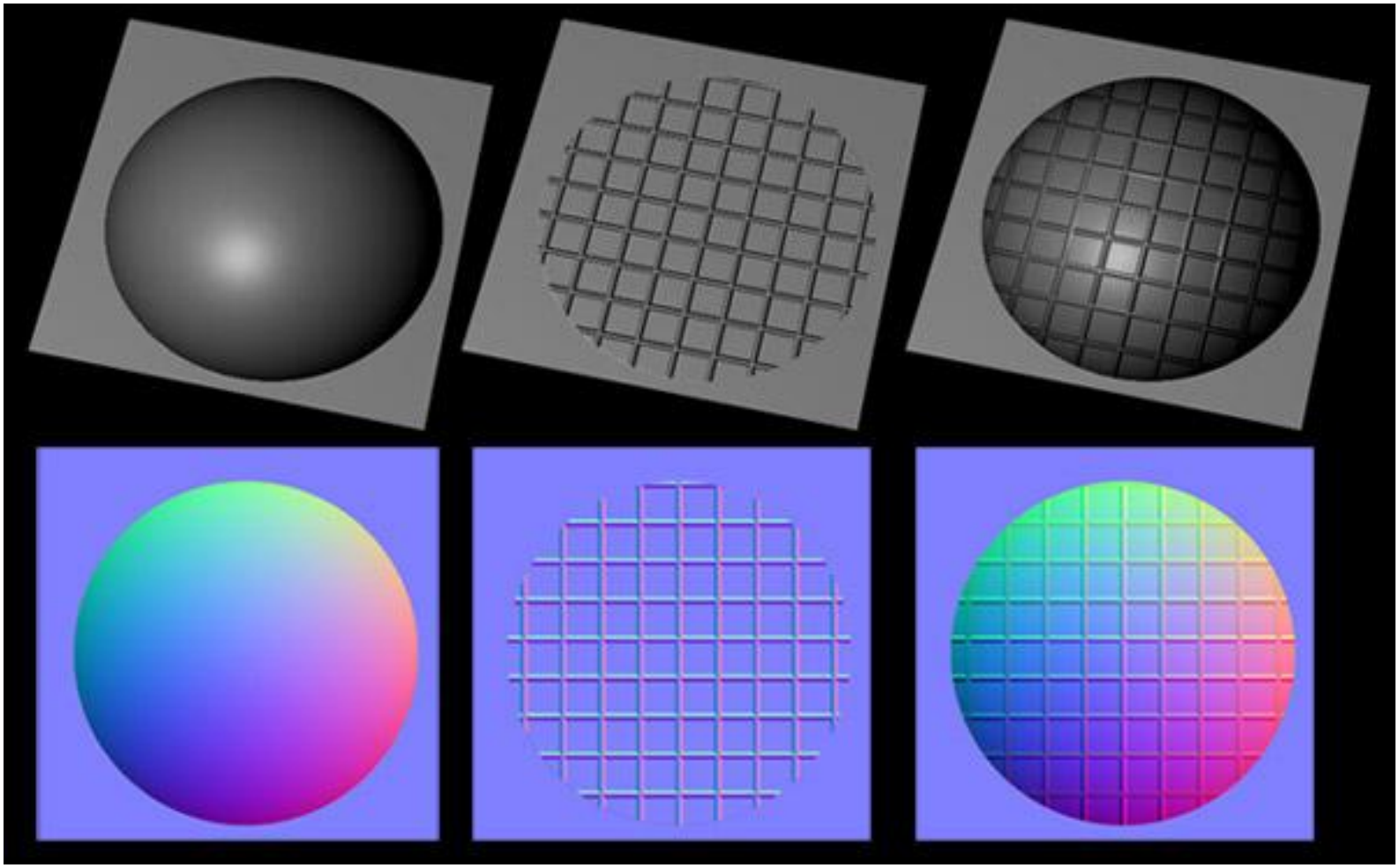
Bump Mapping WTF?

- Au pixel shader on récupère la couleur de la NormalMap que l'on considère comme un vecteur
- On transforme ensuite ce vecteur avec la matrice formée par les 3 vecteurs (Tangente, Binormal, Normal)
- On utilise la normal transformée pour faire les calculs d'éclairage.

Bump map Image 1



Bump map Image 2



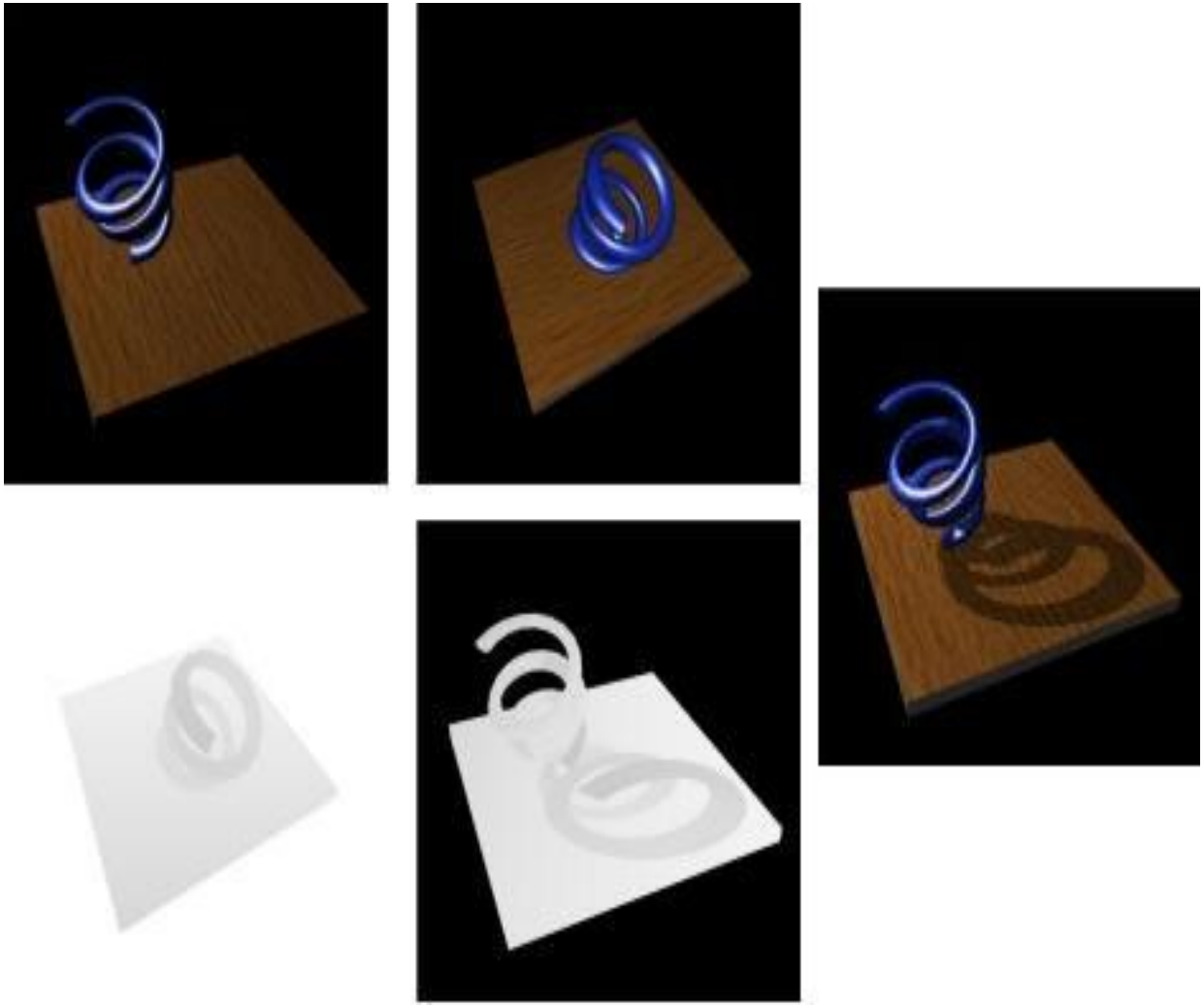
Shadow Mapping

- Chaque lumière qui cast des shadows à une matrice de projection et une ShadowMap
- Cast (du verbe caster ;): qui fait de l'ombre, on est pas encore capable de faire en sorte que toutes les lumières fassent de l'ombre
- ShadowMap : Texture de profondeur

Shadows map algorithms

- L'algorithme :
  - On rend les ShadowsMap
    - On affiche la scène du point de vue la lumière et on enregistre la profondeur de chaque pixel
  - On rend la scène du point de vue de la camera et quand on veut connaitre l'éclairage on fait:
    - On transforme le point dans le repère de la lumière (on utilise la même matrice que pour le rendu de la shadow map)
    - Cela nous donne des coordonnées de texture
    - On regarde la valeur de la shadows map à ces coordonées, ce qui correspond à la profondeur la plus proche
    - Si cette valeur est plus petite que la profondeur du pixel en cours alors c'est ombré

Shadow map en images



Post Process

- Les Post Process sont des rendus effectués après le rendu de la scène.
- En général il s'agit de faire un traitement sur le rendu de la scène une fois celui-ci effectué
- Peut être comparer à des filtres photoshop

Un petit exemple

- On veut faire un rendu en noir et blanc quand le joueur à plus beaucoup de vie.
  - On rend la scène normalement dans une render target
  - On affiche un quad full screen avec en texture la render target contenant la scène rendu. Les coordonnées d'un quad full screen sont directement en screen space
  - Au pixel shader on lit la couleur dans la texture et on applique la formule pour faire du noir et blanc

Conclusion

- Le prochain cours : Le jeu vidéo
- Des questions ?

Une technique

HDR, Tone mapping, Bloom