# ZBufferToy

Codi de la Pràctica 2 de GiVD: ZBufferToy 2022-23

**Abstract**

Aquesta pràctica té com a objectiu principal adaptar algunes funcionalitats de la pràctica anterior de Raytracing utilitzant mètodes projectius a la GPU (ZBuffer). S'ha realitzat la visualització d'objectes representats mitjançant malles triangulars amb materials lambertians i textures, tant directes com indirectes, per obtenir escenes d'alta fidelitat i realistes. S'ha incorporat un conjunt ampli de llums per a millorar la il·luminació de les escenes i s'ha modificat el codi per a carregar escenes de dades temporals i reals (gizmos). A més, la inclusió d'un pla acotat ha permès aportar un fons d'escena o terra sobre el qual mapejar els objectes.

En segon lloc, també s'aprèn a usar els shaders per programar els models d'il·luminació. Es permet la visualització de l'escena amb diferents shaders actius en temps d'execució, que proporcionen diferents tipus d'il·luminació. Durant aquest procés, s'ha treballat en la transmissió de valors al vertex shader i al fragment shader per aconseguir els efectes d'il·luminació desitjats. A través d'aquests shaders s'han pogut explorar nous efectes visuals com la visió nocturna o la tempesta de Fortnite.

A més de tot això, s'han desenvolupat algunes parts opcionals de la pràctica, com la implementació de tres tipus diferents de llums (spot light, ambient light i directional light), el mapping indirecte de textures i les animacions amb dades temporals.

A través d'aquests objectius i funcionalitats, s'han adquirit coneixement sobre l'ús del ZBuffer i la GPU per a la visualització, permetent obtenir representacions d'escenes d'alta fidelitat i realistes amb materials, textures i una il·luminació variada.

**Features**

A continuació s'indica quines parts s'han fet i qui les ha implementat

- Fase 1
    - Lights
        - [x] Ambient Global
            - Núria Torquet
        - [x] Puntual
            - Núria Torquet
        - [x] Direccional
            - Esther Ruano
        - [x] Spotlight
            - Esther Ruano
    - Materials:
        - [x] Pas a la GPU
            - Pau B. i Pau H.
        - [x] Lectura de fitxers .json
            - Pau B. i Pau H.

    - Shading
        - [x] Color
            - Pau B.
        - [x] Normal
            - Esther Ruano
        - [x] Depth
            - Núria Torquet
        - [x] Phong-Gouraud
            - Pau B.
        - [x] Phong-Phong
            - Núria Torquet
        - [x] BlinnPhong-Gouraud
            - Pau B.
        - [x] BlinnPhong-Phong
            - Núria Torquet
        - [x] Cel-shading
            - Pau H
    - Textures
        - [x] Textura com material en un objecte
            - Esther Ruano
        - [x] Textura al pla base
            - Pau B.
    - Adaptació a la lectura de fitxers de dades
        - [x] Escenes virtuals
            - Esther Ruano
        - [x] Escenes de dades
            - Pau B.

- Fase 2

    - [x] Visió nocturna
        - Núria Torquet
    - [x] La Tempesta de Fornite
        - Esther Ruano

- Opcionals

    - [ ] Èmfasi de siluetes
    - [x] Mapping indirecte de textures
        - Esther Ruano
    - [x] Animacions amb dades temporals
        - Núria Torquet
    - [ ] Normal mapping
    - [ ] Entorn amb textures
    - [ ] Reflexions
    - [ ] Transparències via objectes:
    - [ ] Transparències via environmental mapping

**Decisions a destacar**

**Preguntes**

- Pas 1.1: Pas de la llum ambient global a la GPU
    - **Decideix el moment en el que cal passar la llum ambient global a la GPU (al initializeGL? Al updateGL? En crear un objecte?**

      S'ha passat la llum ambient global a la GPU durant la fase d'inicialització (initializeGL) de l'aplicació, ja que la llum ambient global no canvia durant l'execució de l'aplicació. Així es garanteix que la GPU tingui les dades necessàries per renderitzar l'escena amb precisió des del principi. A més, si actualitzes la llum ambient global durant l'execució de l'aplicació, es pot cridar el mètode setAmbientGlobalToGPU de nou per actualitzar els valors a la GPU.

- Pas 1.2: Pas de la llum de tipus puntual a la GPU. Creació de nous tipus de llums
    - **Ara es crea una llum puntual al initializeGL() de la classe GLWidget, quan l'hauries de passar a la GPU? A l'inici de tot? Cada vegada que es visualitza l'escena?**

      En aquest cas la llum puntual s'hauria de passar a la GPU cada vegada que es visualitza l'escena. Això és degut a que la llum puntual pot ser afectada per la posició de la càmera i dels objectes en l'escena, i per tant, la seva representació a la GPU ha de ser actualitzada en cada iteració del bucle de renderitzat.

    - **Què contindrà el "struct" de la GPU? Com l'estructurareu?**

      L'struct de la GPU s'ha definit de la següent manera:

      ```
      struct Light {
      vec3 Ia;
      vec3 Id;
      vec3 Is;

      // Directional lights
      vec3 direction;
      float intensity;

      // spot lights
      vec3 spotDirection;
      float spotCosineCutoff;
      float spotExponent;

      // point lights
      vec3 position;
      vec3 coeficients;
      };
      ```

      S'utilitza un "struct" per a cada llum, on es guarden les seves propietats, com ara la posició, la direcció, l'atenuació... Aquestes estructures s'agrupen en un vector, que es passarà a la GPU per a ser utilitzat en la shader de il·luminació.

- Pas 2.1: Modificació de la classe Material
    - **Utilitzarem també “structs” per a estructurar la informació tant a la CPU com a la GPU, tal i com fèiem a les llums. Des d’on es cridarà aquest mètode? (fent referència al toGPU)** En aquest cas, l'struct del Material és:

```
struct Material
{
    vec3 Ka;
    vec3 Kd;
    vec3 Ks;
    vec3 Kt;
    float shininess;
    float opacity;
};
```

    El `toGPU()` de `GPUMaterial` s'haurà de cridar a la funció draw() dels objectes. Això es deu a que volem que per cada objecte es tingui la informació del seu material. Si ho fessim tots seguits i després els dibuixéssim, aleshores només es tindria a la memòria de la GPU el material de l'últim objecte. Posant-ho al draw aconseguim que abans de pintar cada objecte, es passa el seu material.
    - **Si vols utilitzar diferents shaders en temps d'execució raona on s'inicialitzaran els shaders i com controlar quin shader s'usa? Cal tornar a passar l'escena a la GPU quan es canvia de shader?** Hem canviat la variable program de GLWidget per un vector de GLShaders, i hem afegit un atribut program a GLShader. També hem afegit un atribut `currentShader` que és un enter i guarda la posició del shader actual dins del vector. Podríem guardar un shader o un program directament, però així es gasta menys memmòria. Llavors inicialitzem tots els shaders al principi, al mètode `initShaders()` i així ja els tenim compilats i llestos per ser usats (serà més ràpid canviar de shaders). Llavors per canviar de shader, cridem la funció `updateShader()` després de canviar el `currentShader`. Aqudst mètode activa el shader seleccionat i envia els dades a GPU. (Tornar a enviar l'escena és necessari, ja que al carregar el nou shader sha perdut la informació que ja hi havia.

- Pas 3.1: Creació de diferents tipus de shadings
    - **Gouraud: Fixa't que quan es llegeix un objecte, cada vèrtex ja té la seva normal. Com serà aquest valor de la normal? Uniform o no uniform?** La normal no pot ser de tipus uniform, ja que cada punt de la superfície en té una de diferent. Per això la definim com

```
    layout (location = 1) in vec4 vNormal;
```

    - **En la classe Camera utilitza el mètode toGPU per a passar l'observador als shaders per a que es passi la posició de l'observador cada vegada que s'actualitza la posició de la càmera amb el ratolí. Com serà aquesta variable al shader? Uniform? O IN?**

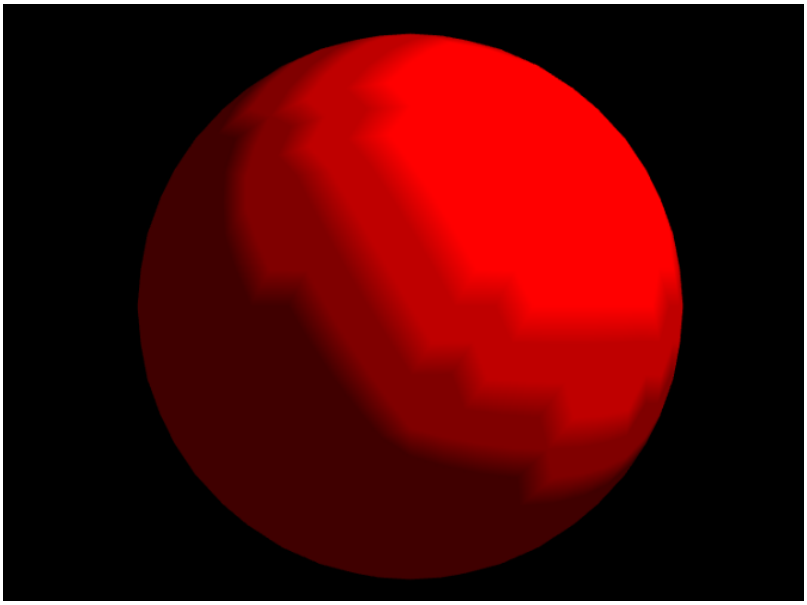
```
uniform vec4 obs;
```

S'ha definit de tipus uniform, perquè no varia el seu valor d'un shader a un altre en la mateixa crida de rendering. El seu valor és uniforme en totes les invocacions.

- **Si vols utilitzar diferents shaders en temps d'execució raona on s'inicialitzaran els shaders i com controlar quin shader s'usa? Cal tornar a passar l'escena a la GPU quan es canvia de shader? I també la càmera?** Hem canviat la variable program de GLWidget per un vector de GLShaders, i hem afegit un atribut program a GLShader. També hem afegit un atribut `currentShader` que és un enter i guarda la posició del shader actual dins del vector. Podríem guardar un shader o un program directament, però així es gasta menys memmòria. Llavors inicialitzem tots els shaders al principi, al mètode `initShaders()` i així ja els tenim compilats i llestos per ser usats (serà més ràpid canviar de shaders). Llavors per canviar de shader, cridem la funció `updateShader()` després de canviar el `currentShader`. Aquest mètode activa el shader seleccionat i envia els dades a GPU. (Tornar a enviar l'escena és necessari, ja que al carregar el nou shader sha perdut la informació que ja hi havia.

- **Quina diferència hi ha entre el Phong-shading i el Gouraud-shading? On l'has de codificar? Necessites uns nous vertex-shader i fragment-shader? Raona on es calcula la il·luminació i modifica convenientment els fitxers de la pràctica.** La diferència principal és que els shaders de Gouraud es calculen al vertex shader i llavors els fragments prenen valors interpolats. En canvi, en el Phong Shader, es calcula el valor per cada fragment, i per això s'ha de fer en el fragment shader. En tots el casos, per fer noves estratègies de shader es necessiten nous fitxers vertex-shader i fragment-shader.

- **Cel-shading: On s'implementarà el càlcul del color per a tenir més trencament entre las games de colors? Necessites uns nous vertex-shader i fragment-shader? Raona on es calcula la il·luminació** El càlcul de color s'implementa en el fragment shader per a que siguin més marcades les diferents franges. En cas que es calculés el color en el vertex shader el resultat seria el següent:



Es necessiten nous vertex i fragment shaders. En el primer cas, perquè es produeixen una serie de càlculs (productes escalars entre la posició de la llum i la de la càmera) que dona com a resultat la variable alpha, que es passa al fragment shader. També és necessari un nou en aquest cas, perquè s'utilitza una estratègia completament diferent a les utilitzades anteriorment i per tant no es pot reutilitzar. Tant la iluminació com la variable alpha es calcula en el vertex shader per tal de reduir el nombre d'operacions a realitzar.

- Pas 4.1: Inclusió de textures
    - Per implementar les textures hem creat un nou material, MATERIALTEXTURA, inspirant-nos en la primera pràctica. Té les mateixes característiques que un lambertian però a més incorpora una Textura. Hem creat de manera simètrica GPUMaterialTextura.

      Per facilitar les comprovacions, a més, desem en un atribut `type` (de la classe GPUMesh) el tipus de material que és segons llegim al mètode `read()`. És en funció d'aquest atribut que s'envia o no el vector de textures, i s'assigna un valor de cert o fals a la variable uniforme `hasTexture`; des del mètode `toGPU()` de GPUMesh.

      Els shaders que permeten veure les textures són Gouraud (Phong i Blinn Phong), Phong i BlinnPhong

- FASE 2:
    - Pas 1.1: Visió Nocturna o Target amb cercle verd
        - **Detalla on es faria el càlcul? Amb quines coordenades? Amb coordenades de món? De càmera? O de viewport?**

          El càlcul es faria a nivell de viewport, utilitzant les coordenades de la finestra de visualització (viewport). Aquestes coordenades es proporcionen pel gl_fragCoord, que indica en quin píxel s'està pintant l'objecte. Per realitzar el càlcul, es necessiten les següents dades:

            - Mida horitzontal i vertical del viewport: Aquestes dades es passen com a variables uniform al fragment shader. La mida es proporciona en coordenades de píxels de la finestra de visualització i s'utilitza per determinar el radi del cercle de visió.
            - Radi del cercle de visió: També es passa com a variable uniform al fragment shader. El radi es calcula com la meitat de la mida horitzontal o vertical del viewport, ja que es vol que el cercle de visió tingui un radi igual a la meitat de l'amplada del viewport. Amb aquestes dades, es pot calcular la distància entre el píxel actual en el fragment shader (gl_fragCoord) i el centre de l'escena, que està en el punt més allunyat de la capsa contenidora. La distància es calcula utilitzant la fórmula de la distància euclidiana entre dos punts. Finalment, es compara la distància amb el radi del cercle de visió. Si la distància és inferior o igual al radi, significa que el píxel està dins del cercle de visió i s'ha de pintar de color verd. En cas contrari, el píxel es pinta de color negre.

        - **Com aconseguiries que els píxels de fons inclosos en el cercle de visió nocturna es pintessin també de color verd?**

Per aconseguir que els píxels de fons inclosos en el cercle de visió nocturna es pintin de color verd, es pot afegir un pla en el punt més allunyat de la capsa contenidora de l'escena. Aquest pla s'anomenaria "pla de fons" i serviria com a superfície per als píxels de fons que estan més enllà dels objectes de l'escena. Aquest pla ha de ser perpendicular al vector de càmera per assegurar que cobreix tot el fons de l'escena.

- Pas 1.2: La tempesta de Fornite
  - **Considera quants parells de vèrtex-fragment shaders has d'usar, a on cal considerar el test amb l'esfera, etc.**

    Farem servir tres parells de vèrtex-fragment shaders: BlinnPhong pels objectes completament exteriors, una modificació de Gouraud (que posa el valor de blau de la component difusa a 1) pels objectes interiors i un de mixt que en funció de la distància de cada vèrtex al centre de l'esfera calcula Gouraud blavós o Blinn Phong.

    El radi de l'esfera es decideix al mètode calculaRadi() de la classe GPUScene, i hem decidit que sigui el mínim entre 1/3 del diàmetre de l'escena i 90.

    És GLWidget qui activa cada un dels shaders i envia els elements de dins, fora i interesectas amb l'esfera fent servir tres mètodes toGPU creats a GPUScene especialment per això: `toGPUIn(), toGPUOut() i toGPUIntersect()`

**Extensions addicionals**

- Tipus de llums: s'han inclòs tres tipus de llums: directional light, spot light i ambient light. A més, per a que l'usuari pugui interactuar ràpidament amb l'escena s'han afegit noves pestanyes a la ui per a modificar els valors de les llums. Ara bé, aquests paràmetres únicament es modificaran si prèviament l'usuari ha introduït la llum al vector de llums de l'escena a través de la calsse GLWidget. Als shaders es té en compte el tipus de llum amb els que s'està treballant, i computarà el color tenint en compte els paràmetres de cada llum en concret.
- Textures indirectes: hem fet el càlcul del mapejat a CPU, no GPU, en funció de la variable indirectMapping que llegim dels arxius JSON i de la qual en suposem un valor fals per defecte.
- Pla afitat al shader night-vision: s'ha utilitzat el pla afitat implementat en el pas 5 de la fase 1 per a simular un fons en l'escena. S'ha calculat la seva normal a través del vector director de la càmera. I la seva posició amb el càlcul de la capsa mínima contenidora a GPUScene. En el cas que l'usuari seleccioni múltiples vegades l'activació del night-shader, únicament es generarà un pla de fons, eliminant el que hi havia prèviament.
- Animacions de dades temporals: s'ha implementat l'animació de malles poligonals afegint les transformacions `translateTG` i `scaleTG` als vèrtex de la GPUMesh.

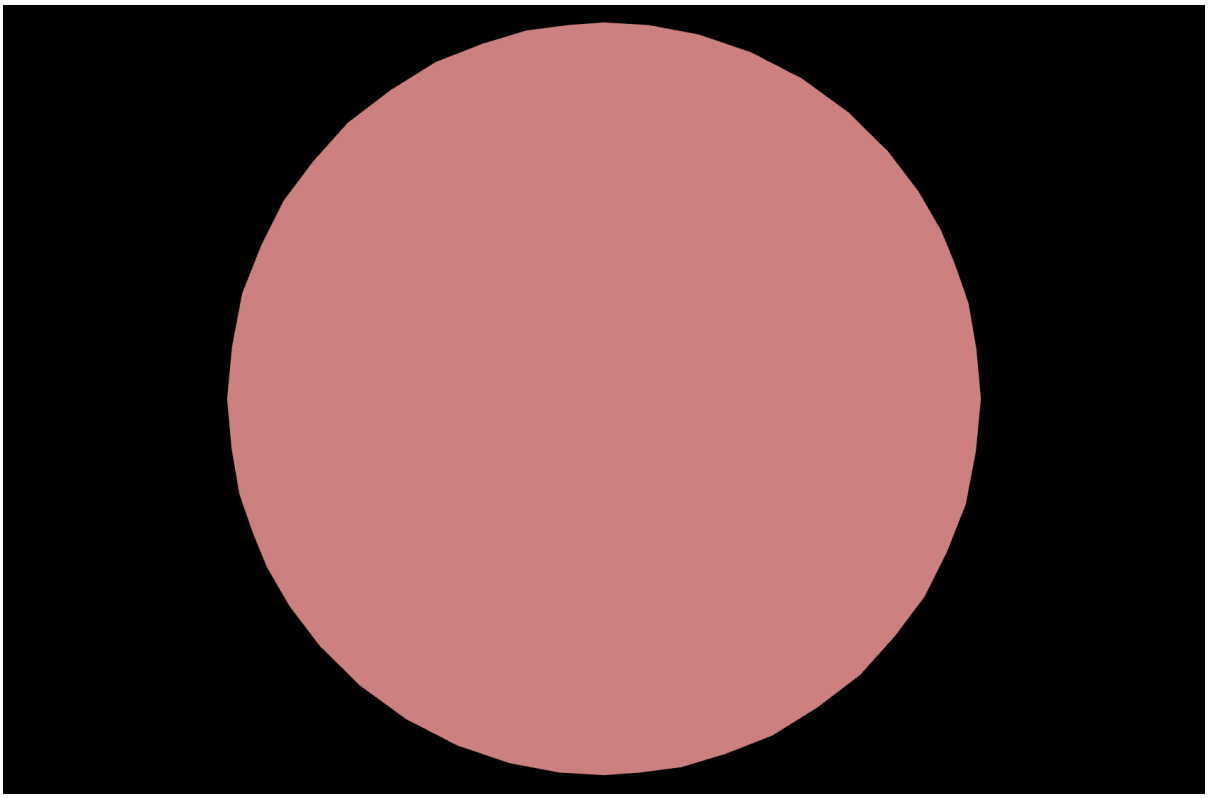
**Screenshots de cada part**

- Pas 3.1: Creació de diferents tipus de shadings

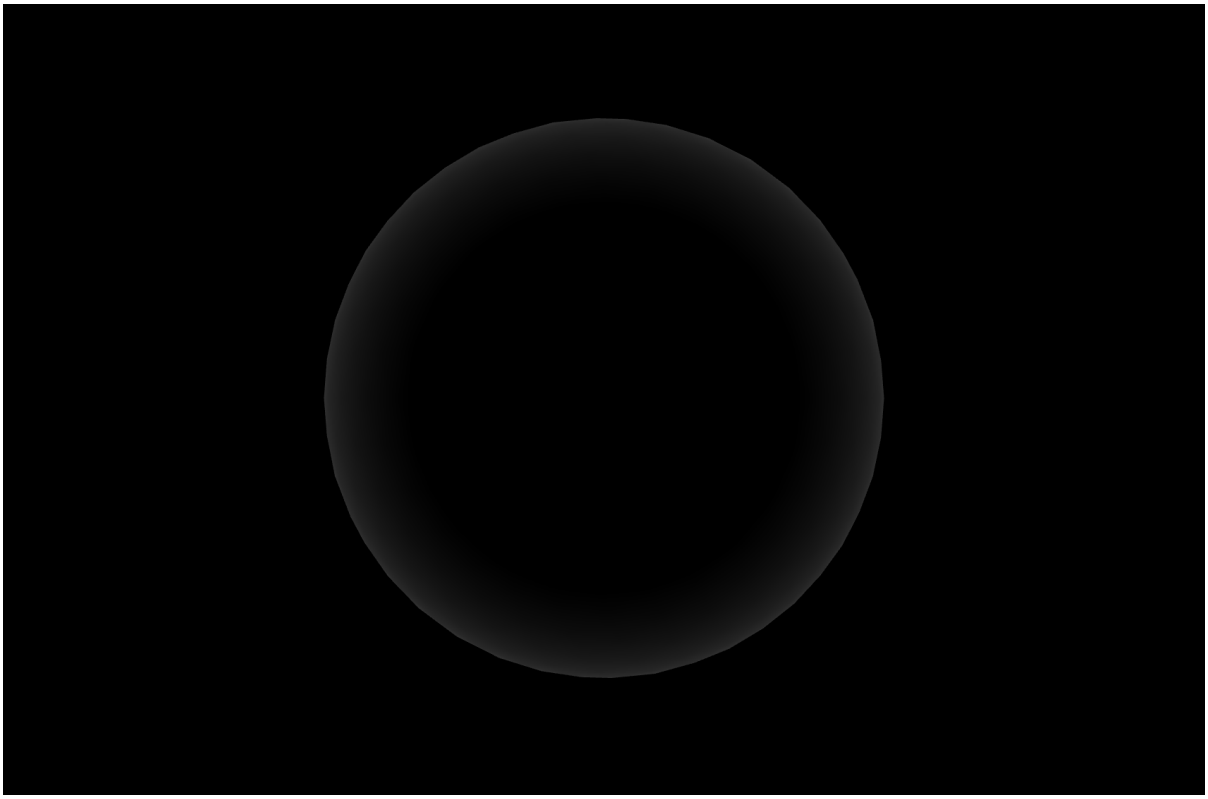
  S'ha definit una GPULight de tipus PointLight per a fer les proves de visualització. Els paràmetres per a inicialitzar-la són els següents:

```
// Default point light
vec3 position1 = vec3(-25,25,25);
vec3 Ia1 = vec3(0.3,0.3,0.3);
vec3 Id1 = vec3(1,1,1);
vec3 Is1 = vec3(0.5,0.5,0.5);
float a1 = 0.0;
float b1 = 0.0;
float c1 = 1.0;
```

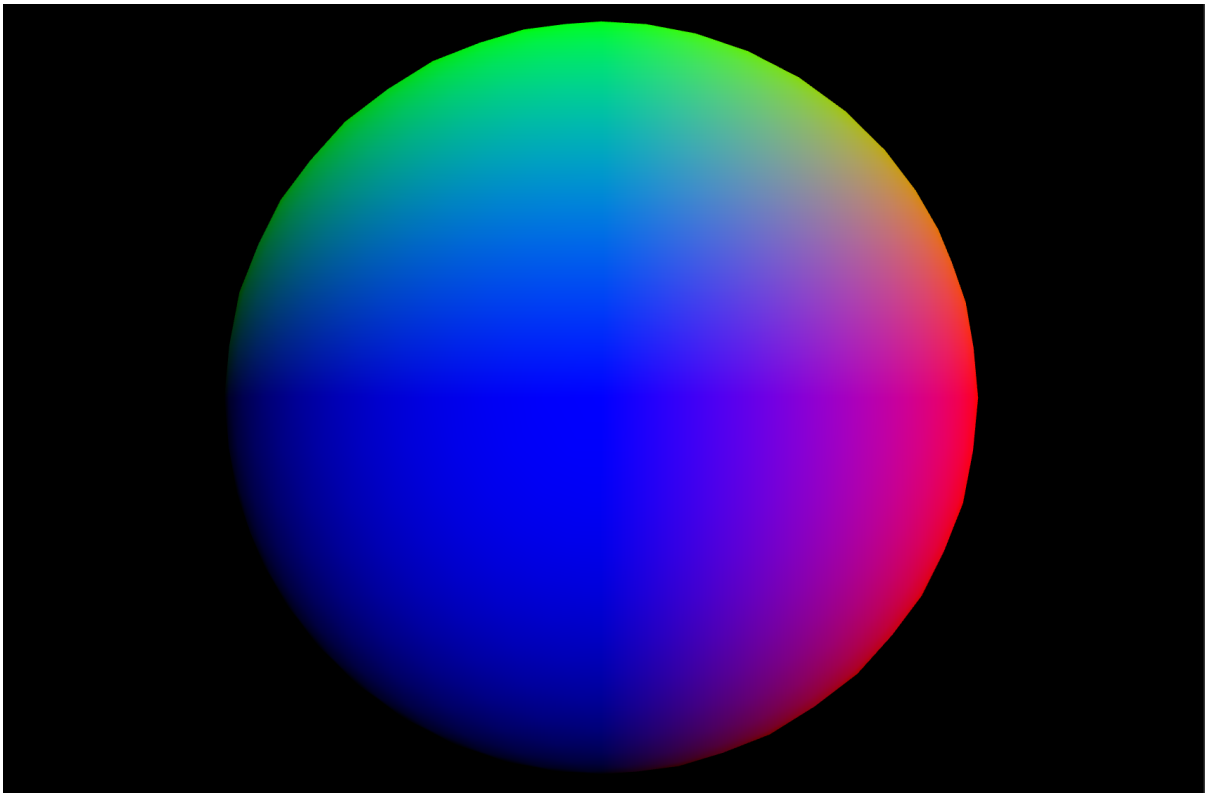
- Color shading



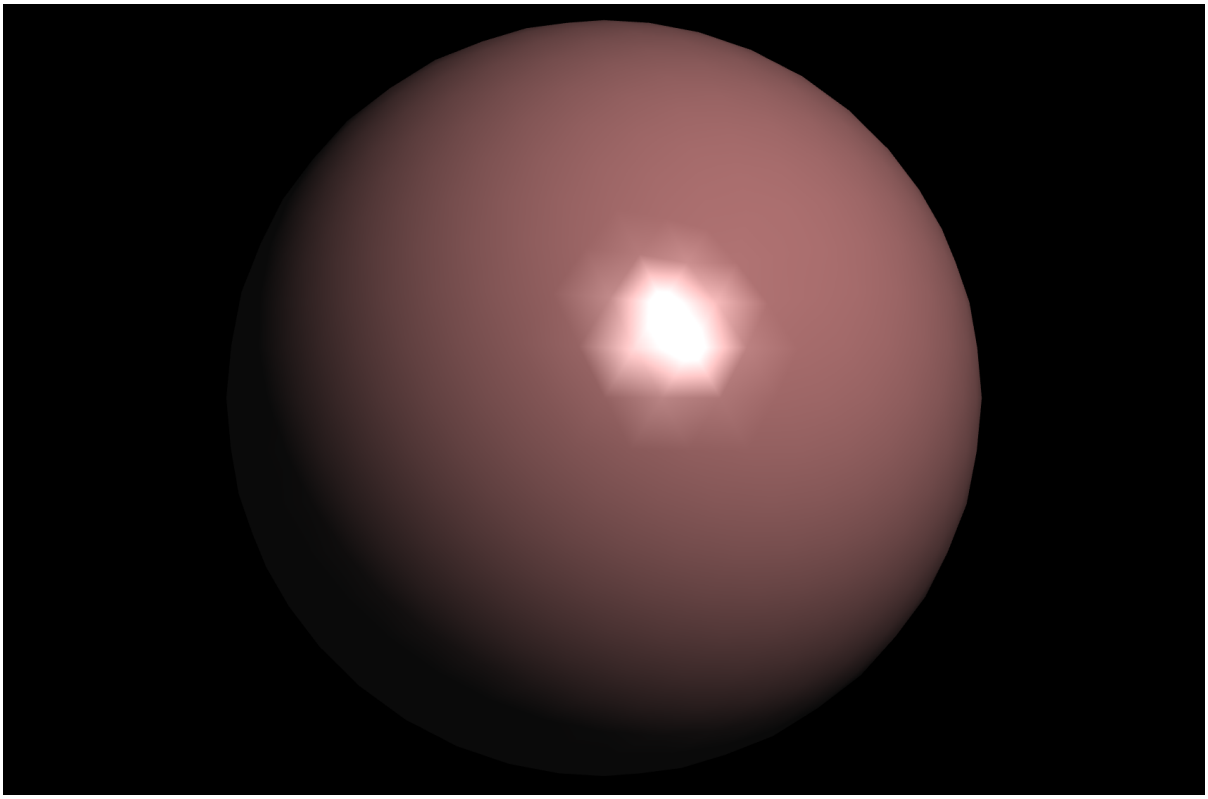
- Depth shading



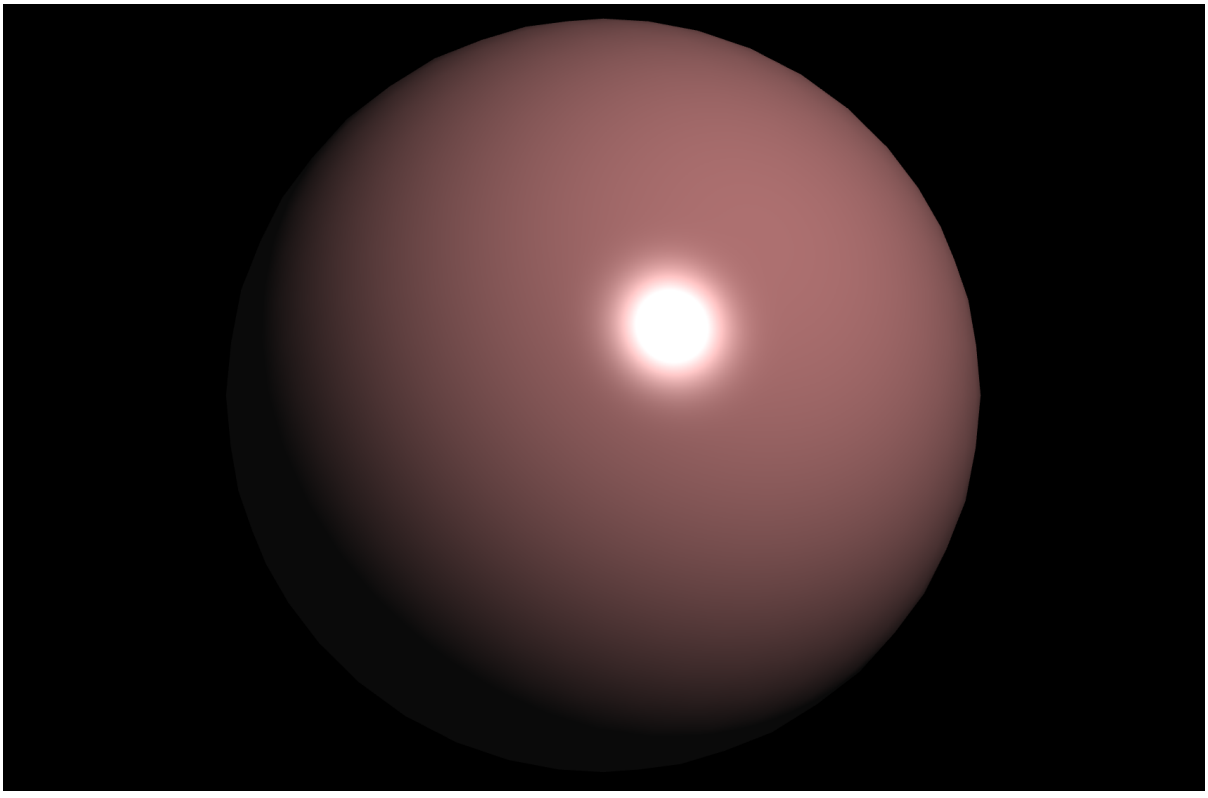
- Normal shading



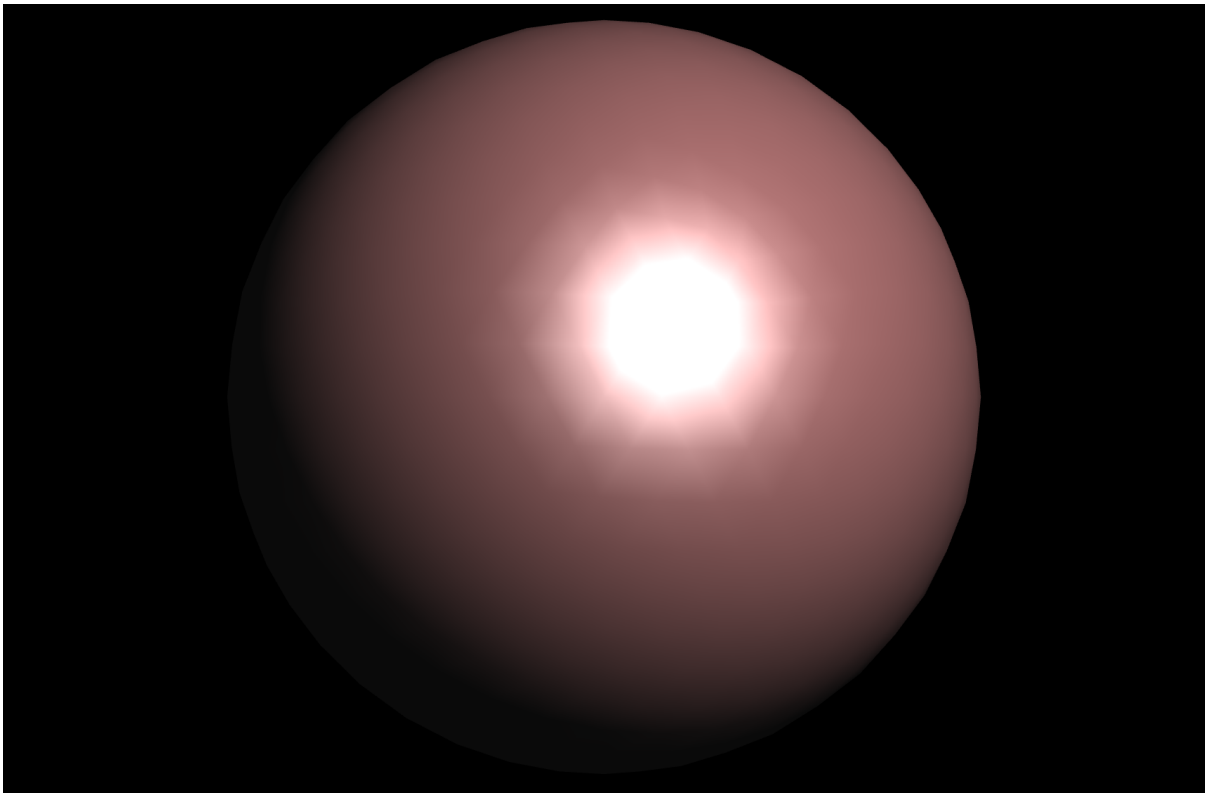
- Gouraud-phong shading



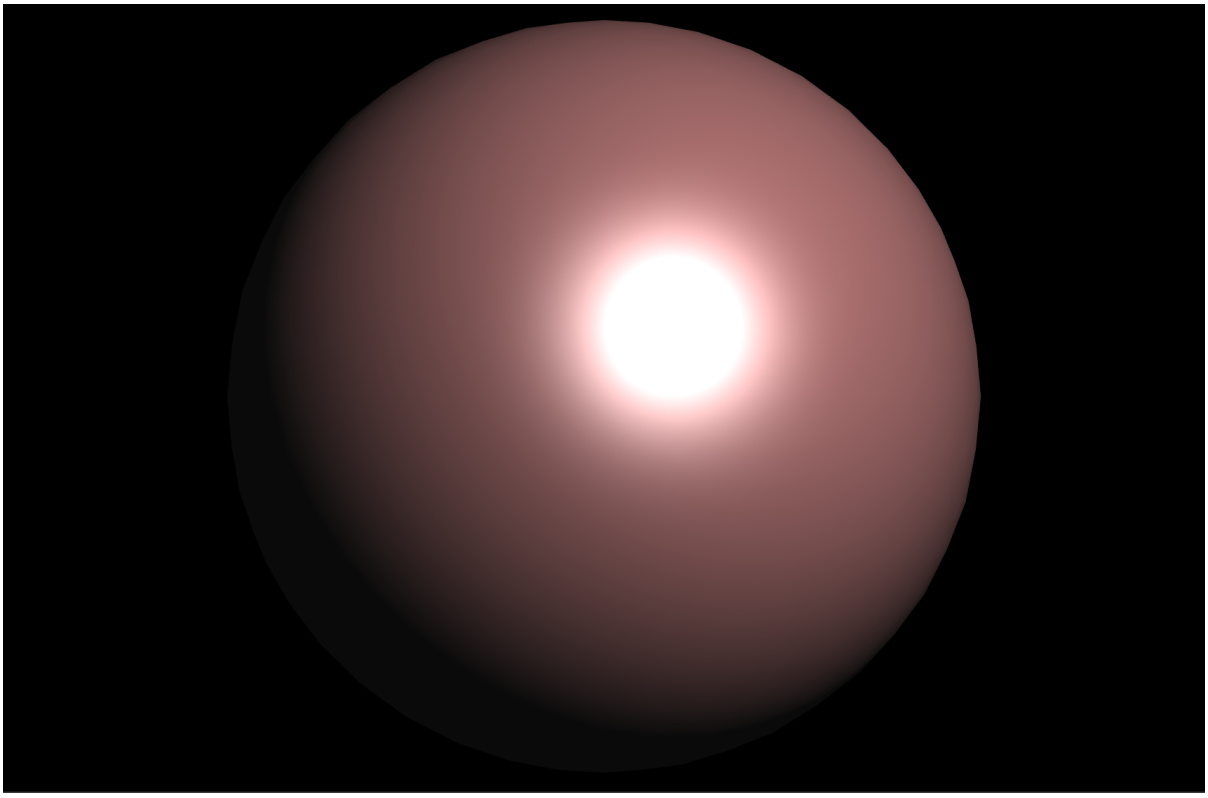
- Phong-phong shading



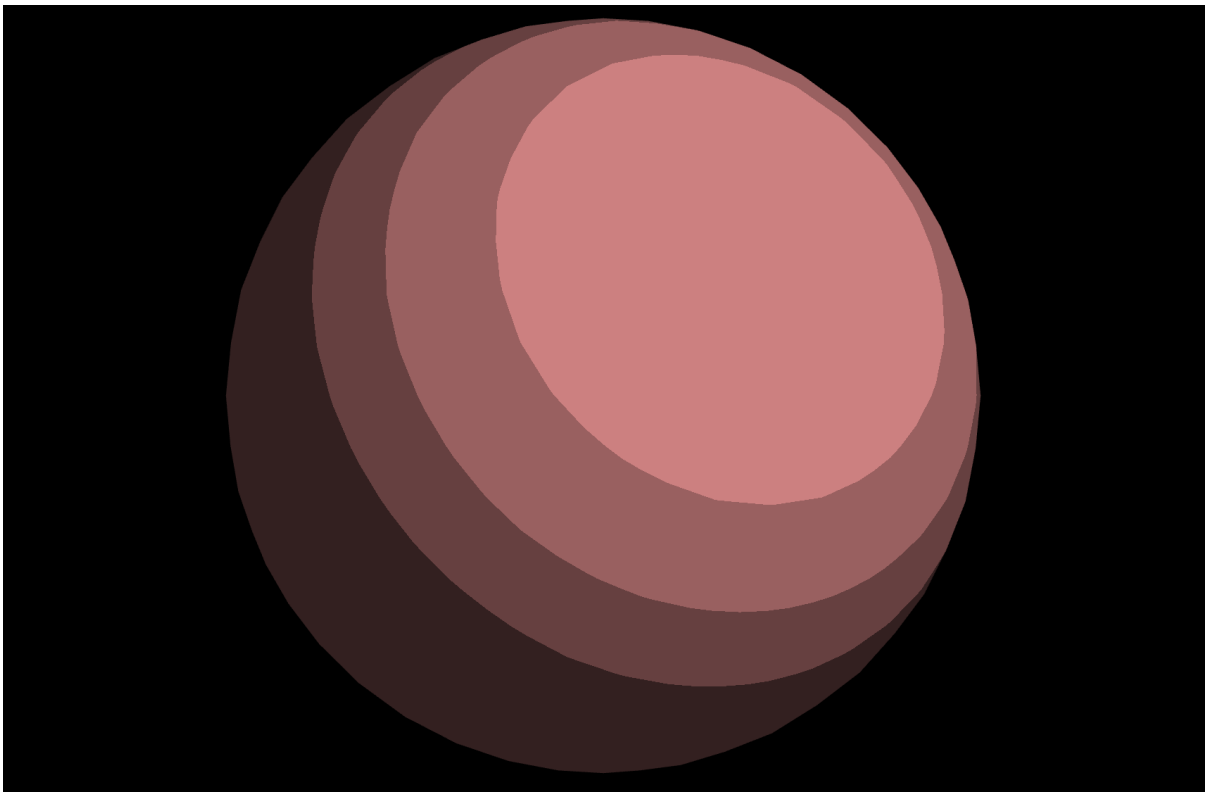
- Gouraud-blinn-phong shading



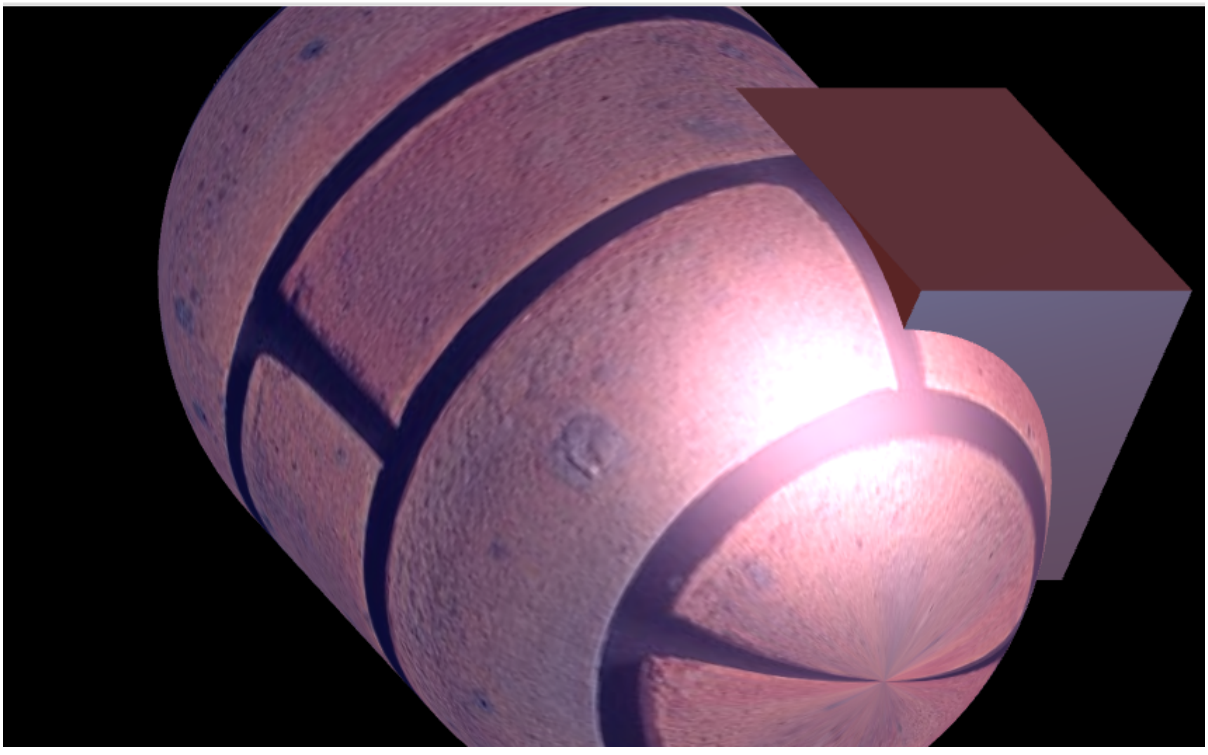
- Phong-blinn-phong shading



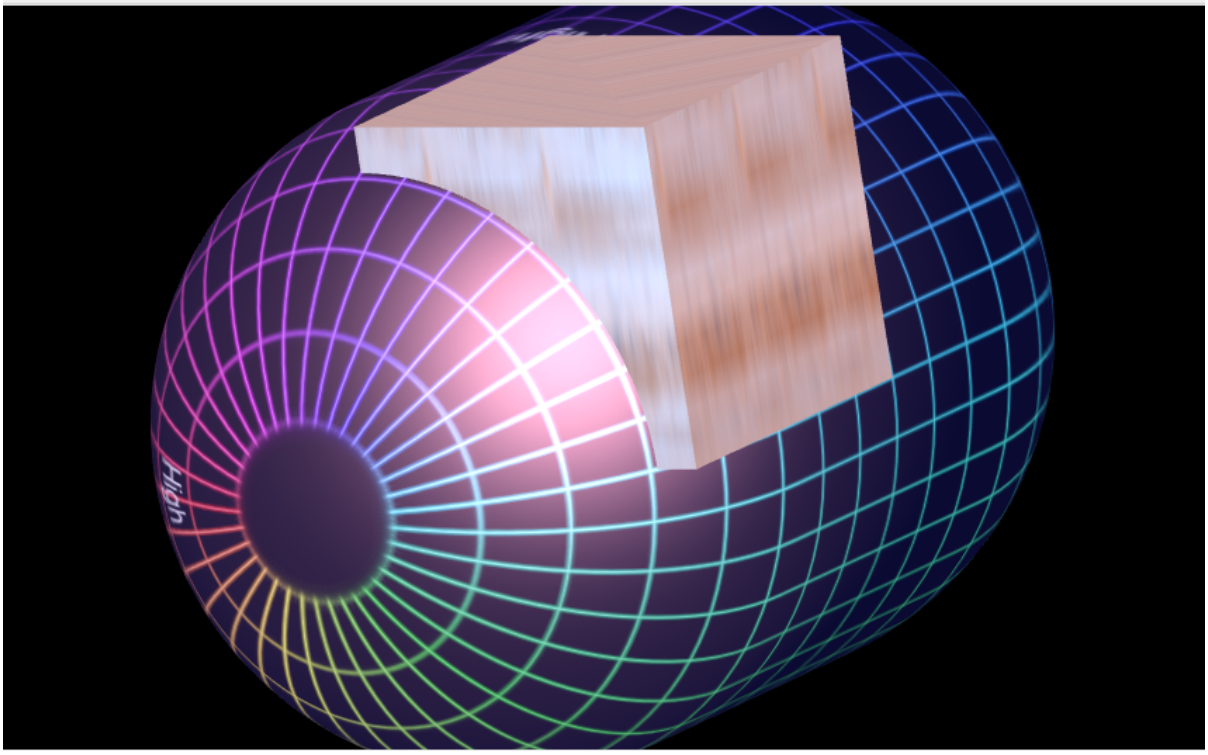
- Cell shading



- Pas 4: Inclusió de textures
    - Textures directes (meshWithTextures.json)



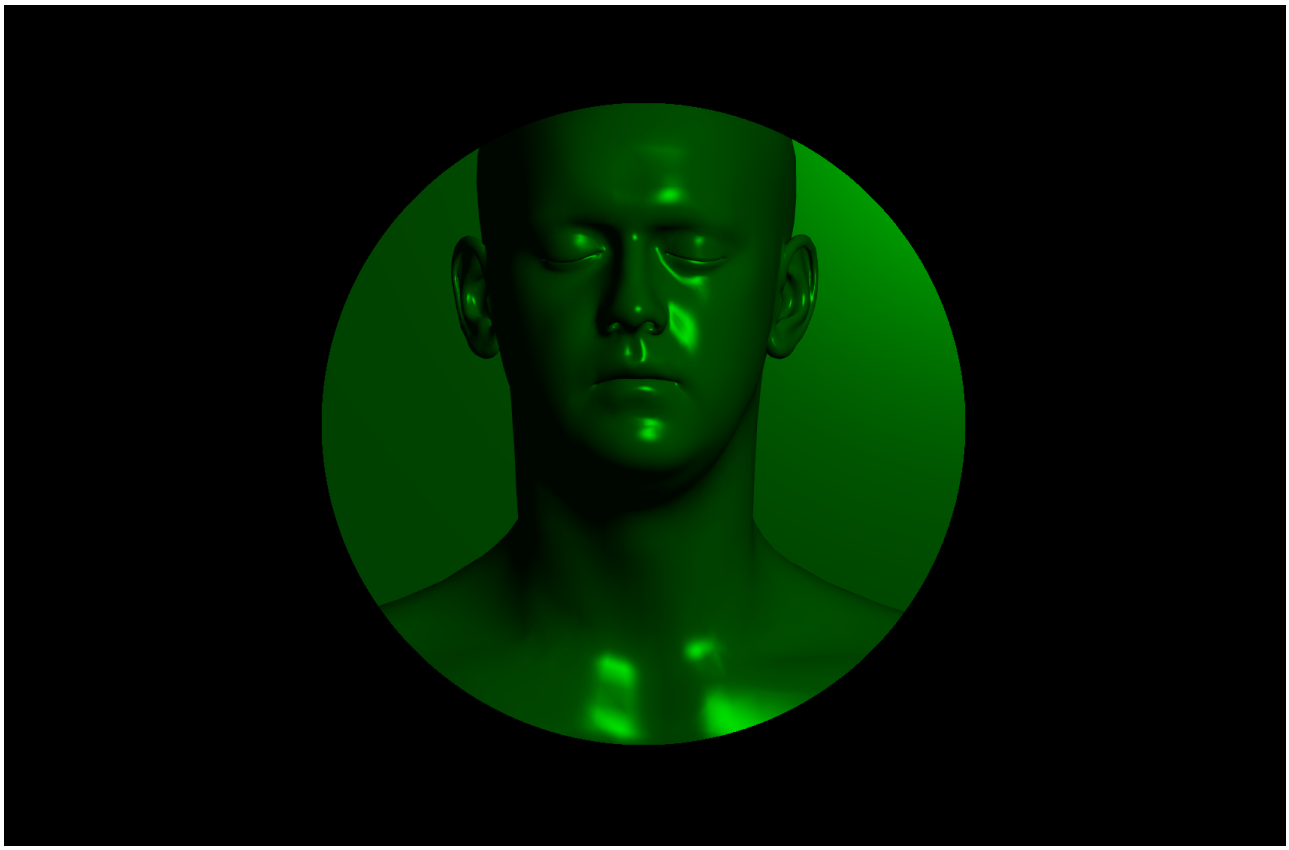
    - [opcional] Textures indirectes (meshWithIndirectTextures.json) Notem que hi ha una esfera dins la càpsula per comprovar que amb el Fortnite shader es carreguen correctament els objectes de dins la tempesta



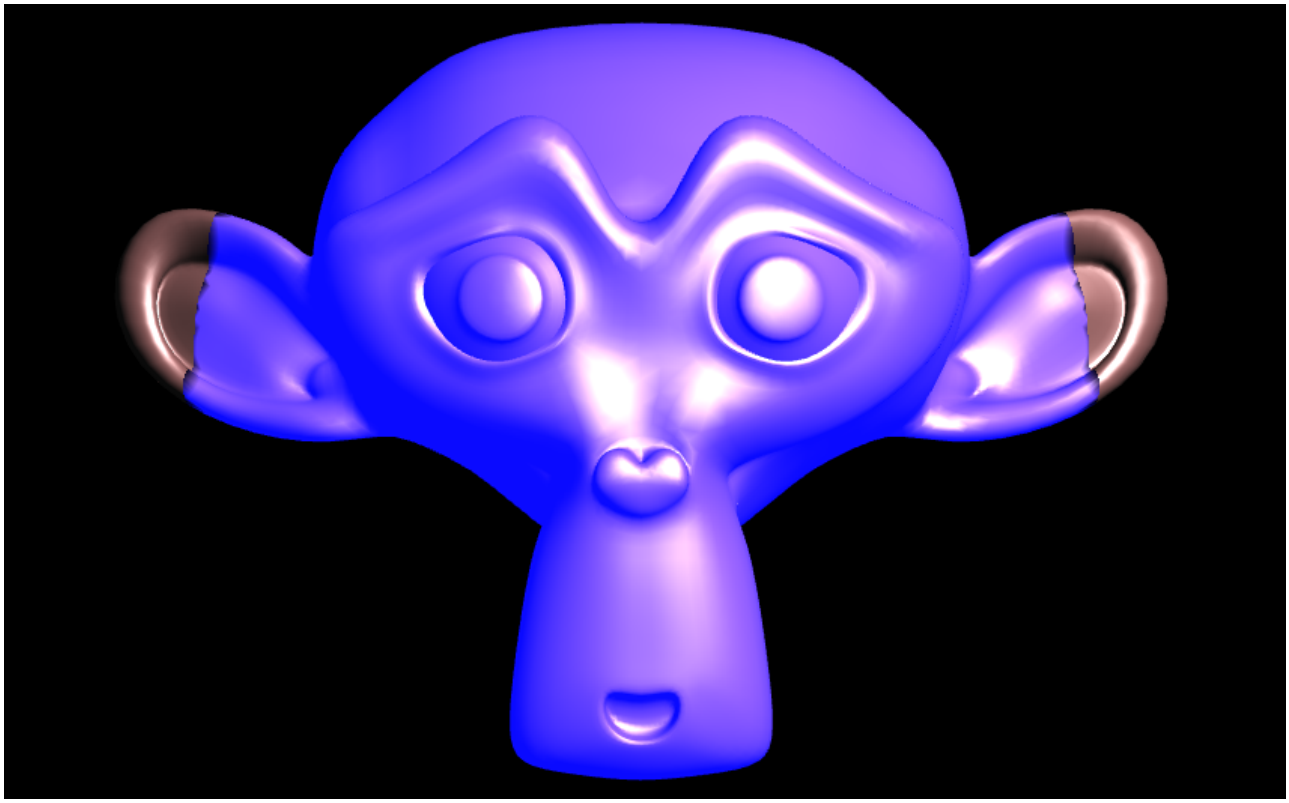

- Pas 5: Dades geolocalitzades
    - Textures directes (data10.json)

    - Textures a les dades geolocalitzades (data10MapBCN.json)


- Pas 2.1: Visió Nocturna o Target amb cercle verd



- Pas 2.2: La tempesta de Fortnite



- [opcional] Animacions amb dades temporals (meshAnimated.json) L'esfera experimenta una escalació amb raó 1.01 durant 11 frames i una translació amb vector de translació (-0.05, -0.05, -0.05) durant 9 frames.