Informe de seguiment I

Kevin Martín Fernández

1 ACTUALITZACIÓ DELS OBJECTIUS

En aquest apartat veurem els objectius complets (Verd) i els objectius que estan començats però no acabats (Taronja).

1. Analitzar
2. Definir
   - 2.1. Definir mòduls interessants pel projecte
   - 2.2. Definir l'estructura del software
   - 2.3. Definir plataformes utilitzades
     - 2.3.1. Definir els mòduls a desenvolupar
     - 2.3.2. Definir la comunicació entre els mòduls
     - 2.3.3. Definir estructura de les dades que rebrà Unreal Engine
3. Desenvolupar
   - 3.1. Desenvolupar mòdul de transformació i obtenció de dades
   - 3.2. Desenvolupar mòdul gràfic (Ureal Engine)
     - 3.2.1. Desenvolupar la interfície del menú
     - 3.2.2. Desenvolupar la lògica del vehicle
     - 3.2.3. Desenvolupar codi per la carrega de terreny y material
     - 3.2.4. Desenvolupar llibreria RPC per el control del entorn
   - 3.3. Desenvolupar mòdul de scripting
     - 3.3.1. Desenvolupament client que controlarà el vehicle
   - 3.4. Integrar els mòduls d'AirSim en Unreal Engine
     - 3.4.1. Integrar mòdul de Segmentació en el projecte
   - 3.5. Desenvolupar altres capes d'informació
4. Testejar
   - 4.1. Fer provés del mòdul de transformació de dades
   - 4.2. Fer provés del mòdul gràfic
   - 4.3. Fer provés del mòdul de control per scripting
     - 4.3.1. Elaborar script d'exemple
     - 4.3.2. Provar script d'exemple
5. Documentar
   - 5.1. Redactar informe inicial
   - 5.2. Redactar informe de seguiment I
   - 5.3. Redactar informe de seguiment II
   - 5.4. Redactar l'informe final
   - 5.5. Elaborar proposta de presentació
   - 5.6. Elaborar pòster
   - 5.7. Gestionar la documentació del dossier

2 MÒDUL GEOTOOLS

Mòdul generat en Python amb l'objectiu d'obtenir informació de mapes d'elevacions, ortofotos, etc. Amb l'objectiu posterior de fer un tractament d'aquestes dades per la generació de terrenys tridimensional i informació en format visible per tal de poder visualitzar-ho en un motor gràfic de tipus Unreal, Unity, OpenGL, etc.

2.1 Obtenció de dades

Amb l'objectiu d'obtenir dades geogràfiques s'ha optat per la comunicació amb els estàndards proposats per l'Open Geospatial Consortium[**?**]: Web Map Service[**?**] encarregat de posar a disponibilitat dades d'imatge com poden ser ortofotos d'una zona geogràfica seleccionada i Web Coverage Service[**?**] encarregat de retornar informació referent a les elevacions

de terreny en una zona geogràfica concreta. Per tal d'obtenir dades es fan peticions HTTP a les direccions web oferides per distintes institucions que segueixen els estàndard anomenats, en aquest cas s'ha realitzat proves amb l'Institut Cartogràfic i Geològic de Catalunya[**?**].

2.1.1 Fitxer de configuració

Per tal de determinar quines dades volem obtenir i de quins webservices l'aplicació accepta per paràmetre un fitxer de configuració en format JSON que ens permet determinar diferents propietats de les dades que demanarem com es pot veure en l'apèndix **??**. En aquest fitxer es pot configurar els següents paràmetres:

- **Type**: Fa referència al tipus de coordenades que li passarem, pot ser latlong o xy en el primer cas farà la corresponent transformació al format UTM (xy)
- **Coordinates**: Coordenades sobre les quals volem fer la petició en el format indicat en el camp type. Si s'escull "xy"es definirà els atributs x, y en cas d'escollir l'latlong"definirem els atributs lat, long.
- **Dimensions**: En aquesta secció escollirem les dimensions que volem que es demanin en les peticions per tal de mantenir la mateixa zona geogràfica.
  - Bbox: Aquestes seran les dimensions de la zona que volem obtenir amb les que es calcularà els límits que delimitaran la zona.
  - Texture: Aquesta serà la resolució que obtindrem per les diferents textures.
- **Wcsurl**: URL al webservice que ens donarà les dades d'altures.
- **Outputwcs**: Nom de sortida del fitxer generat per les altures
- **Formatwcs**: Format del fitxer generat per les altures. Disponibles: raw, obj (Objecte 3D)
- **Wmsrequests**: Array amb cada una de les peticions que farem a diferents WMS per tal d'obtenir vàries imatges
  - Url: URL al webservice WMS
  - Layers: capa o capes que volem obtenir d'aquests webservice
  - Output: Nom del fitxer de sortida
  - OutputFormat: Format del fitxer de sortida. Disponibles: JPG

2.2 Generació de Terrenys a partir dun mapa d'altures

En aquesta secció s'explicarà les diverses formes que s'han optat per generar terrenys que puguin ser interpretats en diferents motors gràfics.

2.2.1 Format RAW

El format RAW és un format pla que es basa en guarda en valors de 16 Bytes totes les altures en format binari tenim com a referència del mar el valor 128, i afegits en el fitxer un darrere de l'altre. Aquest format és acceptat per al creador de terrenys d'Unreal i Unity, però té certes limitacions de dimensions que s'han de complir especificades a l'editor en crear el terreny que fa que es perdi el control de la malla generada, les coordenades de textura no coincideixen amb la textura que es vol aplicar al terreny. Motius pels quals s'ha optat per afegir la generació de l'objecte 3D en format estàndard definit nosaltres l'objecte com es podrà veure en l'apartat **??**, ja que aquestes limitacions fan que es facin vàries adaptacions no estàndards perquè es pugui visualitzar correctament.

2.2.2 Generació de malla 3D

Per tal d'importar terrenys en els motors gràfics s'ha optat per generar una malla 3D en format Wavefront obj[**?**] format compatible amb qualsevol editor 3D, motor gràfic, etc. Aquest format dóna la llibertat per tal de controla la distància entre els vèrtexs, on s'aplicarà la textura i quines seran les normals dels vèrtexs fent que el terreny sigui suavitzat.

Com que el tractament amb bucles és lent s'ha realitzat tots els càlculs amb la llibreria NumPy aprofitant l'eficiència que incorpora aquesta llibreria amb el càlcul de matrius pel qual s'ha adaptat el problema com podem veure en el codi disponible en l'annex **??**.

Per la generació d'objectes cal definir 4 tipus d'objectes:

- **Vèrtex**: Són cada un dels punts en el món, van definits pels índexs segons llegui'm la graella d'elevacions, es multipliquen per un K (Distancia entre els vèrtexs segons la distància que ens indiqui el mapa d'elevacions obtingut).
- **Vèrtex de textura**: Vèrtex amb dos components x, y compresos entre el 0 i 1 que indican la correspondència entre els punts d'una textura i la malla en la qual es vol aplicar aquella textura. Aquestes propietats es calcularien amb

les equacions **??** i **??**.

$$u = f(columna) = columna/(ample - 1) \tag{1}$$

$$v = f(fila) = 1 - (fila/(altura - 1)) \tag{2}$$

- **Normal del vèrtex**: Vectors que ens indica la direcció en la qual es reflecteix la llum per a cada vèrtex de l'objecte. Per tal de calcular aquestes normals cal el pas previ de calcular les normals de cada cara, aquestes no seran introduïdes al fitxer final, ja que aquestes les genera'n els motors per defecte segons l'ordre en el qual indiquem els vèrtexs de les cares com es veurà més endavant.
  - Generació de normals de cares: Per tal de generar les normals d'una cara un cop sapiguem la relació entre les cares se seguirà el patró vist en la figura **??** on seguirem l'equació **??** per al càlcul de la normal de la cara. $\vec{A}$ i, realitzarem el producte vectorial $\vec{C} = \vec{B} * \vec{A}$ i per últim normalitzarem el vector $\vec{Normal} = \frac{\vec{C}}{|\vec{C}|}$.
  - Generació de normals en els vèrtexs: Per tal de generar el vector normal per a cada vèrtex utilitzarem l'estructura que es pot veure en la figura **??** aplicant la següent formula a cada vèrtex on V correspon als vèrtexs i F a les cares de la figura **??**:

$$\vec{NormalV} = \vec{F1} + \vec{F2} + \vec{F3} + \vec{F4} + \vec{F5} + \vec{F6} \tag{3}$$

$$\vec{NormalV} = \frac{\vec{NormalV}}{|\ \vec{NormalV}\ |} \tag{4}$$

- **Cares**: En aquest punt determinarem quina és la unió dels vèrtexs per tal de generar les diferents cares de la malla, en aquesta implementació s'ha decidit per fer triangulació, és a dir, per cada quadrat de la nostra malla generarem 2 cares triangulars. És important generar les cares mantenint l'ordre dels vèrtexs contrari a les agulles del rellotge, d'aquesta forma els motors gràfics determina'n que la normal de la cara apuntarà cap dalt visualitzant correctament la malla 3D.

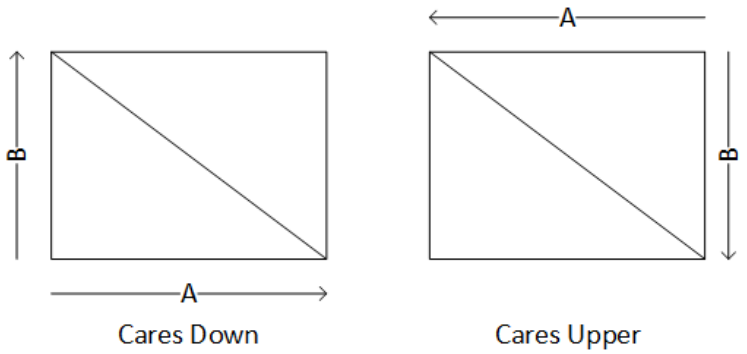

Fig. 1: Patró per càlcul de normals en les cares

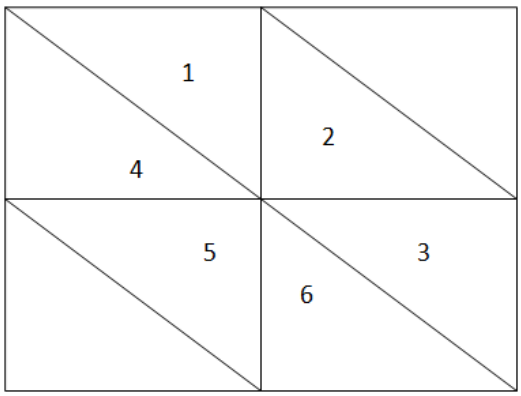

Fig. 2: Patró per càlcul de normals en un vèrtex

Un cop realitzat el procés de generació l'aplicació haurà generat una malla que podem obrir en qualsevol editor com podem veure en la figura **??**

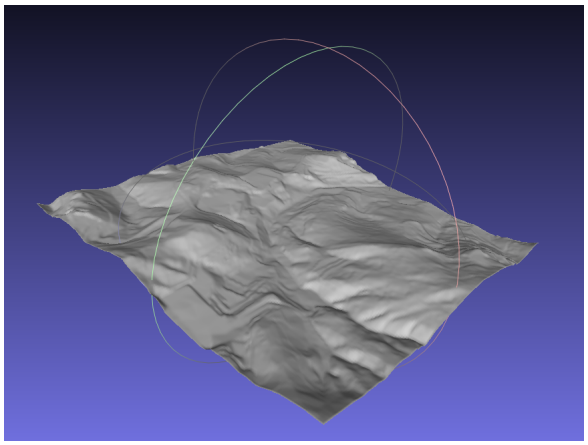

Fig. 3: Malla d'un terreny visualitzada en l'aplicació MeshLab

3 Mòdul: Simulador

En aquesta secció explicarem els diferents punts dels quals es compon el mòdul de simulació amb el que se cerca l'objectiu de posar a disposició una eina que ens permeti simular un viatge amb un vehicle genèric per sobre d'un terreny obtenint imatges des de diferents càmeres configurades anteriorment.

3.1 Vehicle i visualització

Aquesta part serà l'encarregada de donar la interfície per moure un vehicle genèric pel nostre món simulat, aquest vehicle es configurarà amb una sèrie de càmeres (Vegeu més a la secció **??**) que es podran visualitzar en la pantalla i generar imatges de cada una d'elles. Com es pot veure a la figura **??** veurem una vista inicial del nostre vehicle des de la part posterior, i estarem veient en un requadre petit de la nostra pantalla la imatge generada per la càmera inferior.

3.2 Terreny

El terreny és carregat mitjançant codi obtenint la malla i assignant aquesta a un objecte carregat en el món al qual li canviarem el fitxer textura que utilitzara per tal de renderitzar com és veu el terreny.

Aquest material tindrà la capacitat de canviar entre diverses textures per tal de poder carregar informació en diferents espectres i canviar aquesta visualització segons un paràmetre que és podrà canviar des de el client Python.

3.3 Comunicació amb el client d'scripts

Aquesta comunicació es farà utilitzant el protocol RPC, en aquest cas s'utilitza la llibreria Rpclib[**?**] per a C++ per la creació del servidor, la gestió del servidor RPC es fa mitjançant una llibreria estàtica escrita en C++ que s'incorpora al projecte d'Unreal per tal d'abstraure la comunicació RPC amb les crides a codi Unreal.

Referències

[1] Open Geospatial Consortium (OGC) - `http://www.opengeospatial.org/` [08/04/2019]

[2] Web Map Service (WMS) - `https://www.opengeospatial.org/standards/wms` [08/04/2019]

[3] Web Coverage Service (WCS) - `https://www.opengeospatial.org/standards/wcs` [08/04/2019]

[4] Institut Cartogràfic i Geològic de Catalunya (ICGC) - `http://www.icgc.cat/ca/` [08/04/2019]

[5] Wavefront .obj file - `https://en.wikipedia.org/wiki/Wavefront_.obj_file` [09/04/2019]

[6] RPC Lib - Modern mgspack-rpc for C++ - `http://rpclib.net/` [10/04/2019]

Fig. 4: Captura de pantalla del simulador

APÈNDIX

A.1 JSON d'exemple per la configuració de GEOTool

```
{
  "type": "latlong",
  "coordinates" : {
    "lat": 41.4677021,
    "long": 2.2859777
  },
  "dimensions": {
    "bbox":{
      "height": 225,
      "width": 225
    },
    "texture":{
      "height": 1500,
      "width": 1500
    }
  },
  "wcsUrl": "http://geoserveis.icc.cat/icc_mdt/wcs/service",
  "outputWcs": "outputwcs",
  "formatWcs": "obj",
  "wmsRequests": [
    {
      "url": "http://geoserveis.icc.cat/icc_ortohistorica/wms/service",
      "layers": "orto25c2016",
      "output": "outputwms",
      "outputFormat": "jpg"
    }
  ]
}
```

A.2 Codi per la generació d'una malla 3D a partir d'un fitxer d'altures

```
def generate_obj(values, k=5):

    h, w = values.shape

    ij = np.meshgrid(np.arange(h), np.arange(w), indexing='ij')
    i = ij[0].reshape(h*w)
    j = ij[1].reshape(h*w)
    values = values.reshape(h*w)

    # Generate list of vertex
```

```
vertex = np.zeros((h*w, 3))
vertex[:, 0] = i*k
vertex[:, 1] = j*k
vertex[:, 2] = values

# Generate faces
mask = np.logical_and(i < (h-1), j < (w-1))
uFaces = np.zeros((h*w, 3), dtype=np.int32)
indexVertex = np.arange(h*w)+1

uFaces[mask, 0] = indexVertex[mask]
uFaces[mask, 1] = indexVertex[mask] + w
uFaces[mask, 2] = indexVertex[mask] + w + 1
uFaces = uFaces[mask]

dFaces = np.zeros((h * w, 3), dtype=np.int32)
dFaces[mask, 0] = indexVertex[mask] + w + 1
dFaces[mask, 1] = indexVertex[mask] + 1
dFaces[mask, 2] = indexVertex[mask]
dFaces = dFaces[mask]

faces = np.concatenate((uFaces, dFaces), 0)

# Generate UV Map
uvMap = np.zeros((h*w, 2))
uvMap[:, 0] = j / (w - 1)
uvMap[:, 1] = 1 - (i / (h - 1))

# Generate normal faces upperFaces
#mask = np.logical_and(i > 0, j < h-1)
aUpper = (np.roll(vertex, h, axis=0) - vertex)
bUpper = (np.roll(vertex, -1, axis=0) - vertex)

normalUpperFaces = np.cross(bUpper, aUpper)
module = np.linalg.norm(normalUpperFaces, axis=1)
normalUpperFaces[:, 0] = normalUpperFaces[:, 0] / module
normalUpperFaces[:, 1] = normalUpperFaces[:, 1] / module
normalUpperFaces[:, 2] = normalUpperFaces[:, 2] / module

# Generate normal faces downFaces
#mask = np.logical_and(i < w-1, j > 0)
aDown = (np.roll(vertex, -h, axis=0) - vertex)
bDown = (np.roll(vertex, 1, axis=0) - vertex)

normalDownFaces = np.cross(bDown, aDown)
module = np.linalg.norm(normalDownFaces, axis=1)
normalDownFaces[:, 0] = normalDownFaces[:, 0] / module
normalDownFaces[:, 1] = normalDownFaces[:, 1] / module
normalDownFaces[:, 2] = normalDownFaces[:, 2] / module

# Generate vertex normals
normalVertex = np.zeros((h*w, 3))
mask = np.logical_and(np.logical_and(i > 0, j > 0), np.logical_and(i < w-1,
    j < h-1))

normalVertex = np.roll(normalUpperFaces, 1, axis=0)
normalVertex += normalUpperFaces
normalVertex += np.roll(normalUpperFaces, -h, axis=0)
normalVertex += normalDownFaces
normalVertex += np.roll(normalDownFaces, h, axis=0)
normalVertex += np.roll(normalDownFaces, -1, axis=0)
```

```
    module = np.linalg.norm(normalVertex, axis=1)
    valid_modules = module != 0
    normalVertex[valid_modules, 0] = normalVertex[valid_modules, 0] / module[
        valid_modules]
    normalVertex[valid_modules, 1] = normalVertex[valid_modules, 1] / module[
        valid_modules]
    normalVertex[valid_modules, 2] = normalVertex[valid_modules, 2] / module[
        valid_modules]

    normalVertex[np.logical_not(mask), 0] = 0
    normalVertex[np.logical_not(mask), 1] = 0
    normalVertex[np.logical_not(mask), 2] = 1

    return vertex, faces, uvMap, normalVertex
```