UNIVERSIDAD NACIONAL DE AGUSTÍN

Facultad de Ingeniería de Producción y Servicios

Escuela Profesional de Ciencia de la Computación

OCTREE QUANTIZER

Alumno:
Cueva Flores, Jonathan Brandon
Herrera Cooper, Miguel Alexander

Docente:
Machaca Arceda Vicente

Arequipa - Perú

Índice

Índice de figuras

1. Resumen

Muchas de las técnicas de programación utilizadas para resolver problemas bidimensionales se pueden extender a tres dimensiones. Aquí los oct-trees se desarrollan como un análogo tridimensional de quad-trees. Los oct-trees se pueden usar en modelado geométrico y planificación espacial. Se proporciona un algoritmo rápido para la rotación de 90 de las representaciones de objetos en oct-tree. Se proporciona un algoritmo de espacio eficiente para la traducción en el espacio.

2. Introducción

El modelado de objetos tridimensionales y espacios por computadora es importante en la planificación espacial [1], la animación por computadora y la visión artificial . Entre los métodos de representación que se han usado en el pasado se encuentran (1) poliedros descritos por polígonos cuyos vértices se dan como triples coordinados (x, y, z) y (2) una matriz espacial (matriz triplicada con subíndice) cuyos elementos pueden ser 0 o 1, ya que corresponden al espacio vacío o "materia sólida". Se han realizado más investigaciones para los métodos de poliedros porque estos métodos generalmente dan una mayor precisión en la descripción de menos bits de memoria. Existen situaciones computacionales, sin embargo, cuando los enfoques de matriz espacial son más convenientes; operaciones como la intersección son más fáciles de formular para matrices espaciales que para poliedros. Además, las matrices espaciales admiten paralelismo en el procesamiento más fácilmente que los poliedros. En algunos casos, los problemas deben resolverse utilizando un enfoque multinivel donde las soluciones aproximadas se calculan primero utilizando matrices espaciales y luego las soluciones se ajustan con métodos de poliedro. Desarrollamos oct-trees aquí como un medio para hacer que las operaciones de matriz de espacio sean más económicas en términos de espacio de memoria. Oct-trees nos da un método de representación que cae en la categoría recursiva". El enfoque puede considerarse una extensión tridimensional de los métodos de cuatro árboles. Comenzamos definiendo oct-trees. Luego discutimos el complemento, la intersección, la unión y la condensación, y damos algoritmos para calcular rotaciones y traducciones de objetos codificados en oct-tree.

3. Definición

- Una oct-árbol es una estructura de datos para llevar a cabo en tres dimensiones la ubicación del punto s y rango búsquedas.
- En un oct-árbol finito , los datos geométricos se clasifican de una manera que permite búsquedas de elementos eficientes.El espacio que contiene una cuadrícula se descompone recursivamente en subespacios. Un subespacio puede ser un nodo o una hoja terminal.
- Un nodo es un subespacio intermedio que no contiene ningún elemento en sí mismo, pero sirve como contenedor para otros subnodos o subhojas. Una hoja terminal representa una situación en la que una división adicional no reduciría el número de elementos en al menos una de las hojas secundarias resultantes.

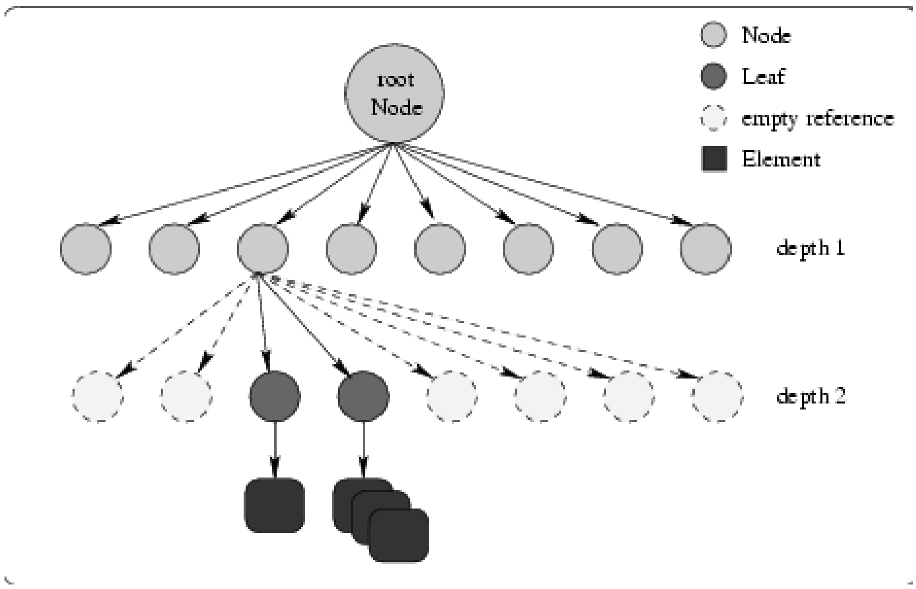


Figura 1: Jerarquía de nodo / hoja de un oct-árbol.

4. Representación

El árbol que presenta al sólido puede ser almacenado de diversas formas.

4.1. Árbol Explícito

Cada nodo almacena su tipo y ocho punteros a sus nodos descendientes. El nodo raız del árbol almacena además las coordenadas de la caja envolvente.

```
struct octree {
     double xmin,ymin,zmin;
     double xmax,ymax,zmax; // Caja envolvente
     struct octree *root;
};

struct octreenode {
     char tipo; // BLANCO, NEGRO o GRIS
     struct octreenode *hijo[8];
};
```

Figura 2: Estructura del Oct-Tree

4.2. Codificación Lineal

Se almacena una lista ordenada del camino a seguir para cada nodo negro. Hay un dıgito octal para cada nivel, de forma que conforme se va leyendo el número de izquierda a derecha el código nos muestra qué hijo tomar para seguir la ruta.

4.3. Codificación Lineal Pre-Orden

Se almacena una lista ordenada de nodos generada al recorrer el árbol en preorden. Para cada nodo se indica su tipo, de forma que tras un nodo gris, aparecerá la codificación de sus ocho hijos.

En general, el número de nodos necesarios para representar un sólido con un octree es proporcional al área de la superficie de dicho objeto [Mea82], por tanto, se puede asegurar que si bien los octrees no necesitan tanto espacio como las enumeraciones exhaustivas, el alcanzar un elevado nivel de detalle supone un coste extra de almacenamiento.

5. Propiedades

- Modelo poderoso, los octrees son modelos de representación aproximada, y pueden ser exactos para ciertos objetos.
- Validez, no requiere de una conectividad especial, todos los octrees son representaciones válidas de algún sólido.
- No ambigüedad y unicidad, hasta los límites de resolución, todos los octrees no ambiguos, definen un sólido.
- Lenguajes de descripción, los octrees son formados mediante una conversión de otras representaciones, como las constructivas; y en el procesamiento de imágenes, los octrees y los quatrees son formados directamente de imágenes digitales de datos, por media de un proceso de "raster".
- Consistencia, en general el número de nodos que un octree representa es proporcional al área del objeto. En promedio un octree fácilmente puede medir más de un millón de bytes de memoria.
- Operaciones cerradas, un octree soporta de algoritmos cerrados para los problemas de translación, rotación y operaciones booleanas.
- Sencillos computacionalmente, muchos algoritmos de los octrees toman la forma de transversal, donde las operaciones son relativamente fáciles para cada nodo del árbol.

6. Extensiones del Oct-Tree

Para subsanar en la medida de lo posible la inexactitud inherente al octree, se han propuesto diversas extensiones al mismo de forma que se pueda representar de forma exacta el objeto sólido. Varias de ellas hace uso de información de la frontera, por lo que pueden ser considerados una aproximación hıbrida.

6.1. Polytrees

Incorpora en los nodos terminales información geométrica de la superficie del objeto, permitiendo una representación del mismo. Esta información se almacena en unos nuevos tipos de nodo, que son: nodos cara, que son aquellos atravesados por una única cara poligonal del sólido; nodos arista, que contienen dos caras adyacentes y parte de su arista común; y nodos vértice que contienen un vértice del poliedro y parte de las caras y aristas que convergen en él. La información de la geometrıa del objeto se calcula y almacena explıcitamente en cada nodo.

6.2. Extended Oct-Trees

Surgieron simultáneamente a los polytrees, y añaden también tres tipos de nodos al esquema clásico del octree: cara, vértice y arista, con idénticas caracterısticas. Sin embargo, la información es calculada y almacenada de forma muy distinta. En los Extended Octrees se almacena en cada nodo la configuración y un conjunto de referencias a una única lista de semiespacios planos, eliminando las redundancias existentes en los polytrees. Por configuración se entiende la información necesaria para clasificar un punto con respecto al objeto. La recursión infinita que también se presenta en los polytrees, que ocurre cuando un nodo es atravesado por un conjunto de caras que convergen en el exterior de nodo, se resuelve añadiendo lo que se denominan nodos grises terminales, que almacenan la configuración del nodo vértice en el que converge dicho conjunto de caras. En la figura antrior se muestran sólidos respresentados con octrees y con Extended Octrees, donde se observa que con éstos últimos son necesarios menos niveles y se consigue una representación exacta.

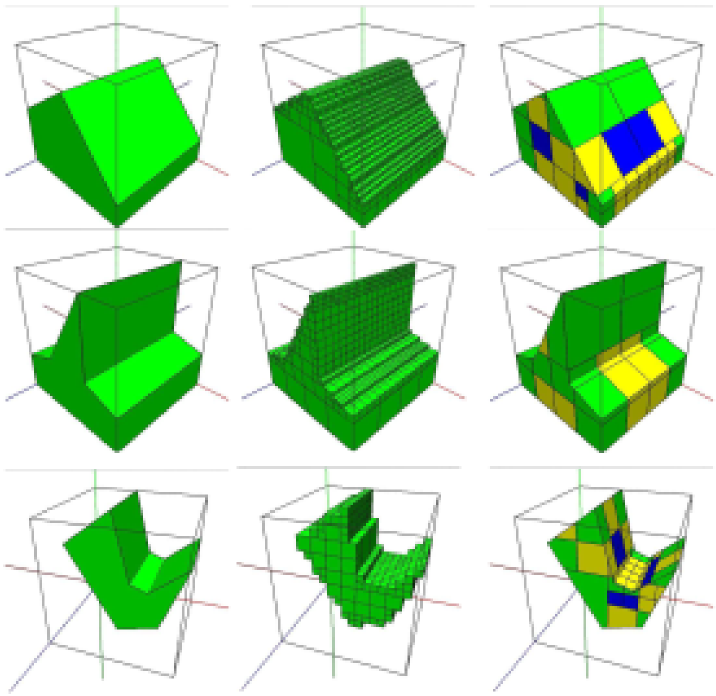


Figura 3: Representación de un sólido (a la izquierda), mediane un octree (columna entral) y un Extended Octree (derecha)

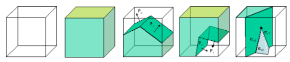


Figura 4: Nodos blanco, negro, convexo, cóncavo y vértice del SP-Octree.

6.3. SP Oct-Tree

(Space Partition Octrees). Esta extensión de los octrees se basa en el uso de semiespacios planos para delimitar el interior y el exterior del sólido, tanto en nodos terminales como en nodos grises intermedios. Cuando un nodo está completamente dentro o fuera del sólido, se clasifica como negro o blanco. A estos dos tipos básicos, el SP-Octree añade los nodos convexos y cóncavos, cuando la intersección del sólido con un nodo del árbol sea un volumen convexo o cóncavo respectivamente. En estos casos, se almacena en el nodo una referencia a los planos que, intersecados con el volumen del nodo, crean el volumen convexo o cóncavo (éste último mediante sustracción de una convexidad al volumen completo). Se incluye también, al igual que en los Extended Octrees el nodo vértice para evitar recursiones infinitas, y el nodo gris, además de indicar una recursión necesaria por contener concavidades y convexidades en la geometrıa abarcada, contiene información de los planos que forman una envolvente convexa de la parte del sólido contenida en el nodo. En la figura 4 podemos ver los distintos tipos de nodo, ası como un sólido representado con esta estructura en la figura 5. Esta estructura permite la realización de forma eficiente de tests de inclusión, ası como una trasmisión progresiva del modelo poliédrico, ya que no almacena la información sólo en los nodos hoja, sino que anticipa la aparición de los planos a los niveles intermedios. Además, supone una reducción en el espacio necesario para el almacenamiento de los modelos.

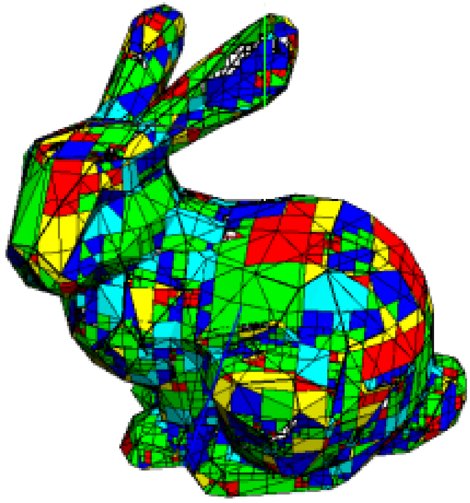


Figura 5: Stanford Bunny representado con SP-Octree. Cada color indica un tipo de nodo.

7. Cuantificación de Partición fija

La cuantificación de color más simple toma una partición de espacio de color fija y predeterminada, y realiza un solo paso sobre la imagen de entrada para asignar un índice de color a cada píxel. El índice de color es siempre se supone que es un índice en tres tablas de colores de 8 bits (RGB). Cada tabla de colores es típicamente 8 bits, para permitir la asignación de color RGB completo en un búfer de cuadro de pantalla de 8 bits. Sin embargo, la memoria recientemente se volvió lo suficientemente económico como para que la mayoría de las pantallas actuales tengan 16 o 24 bits de profundidad. La partición fija de igual volumen sufre de contornos graves y pérdida de fidelidad de color.[1]

8. Aplicación

8.1. Algoritmo de muestreo de importancia de Oct Tree

El algoritmo de muestreo de importancia de oct-tree proporciona un mapeo preciso, eficiente y completo de la ubicación de terremotos PDF s en espacio 3D (xyz).

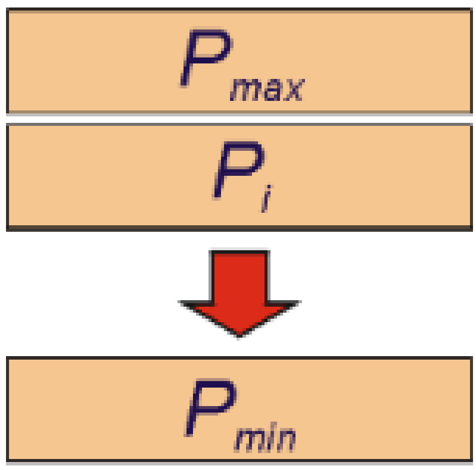
- **Procedimiento** El método oct-tree utiliza la subdivisión recursiva y el muestreo de celdas en el espacio 3D (abajo) para generar una cascada de celdas muestreadas, donde la densidad de las celdas muestreadas sigue los valores PDF del centro de la celda.

- La probabilidad de que la ubicación del terremoto esté en una celda dada i es aproximadamente:

$$P_i = V_i * PDF(x_i)$$

  donde $\mathrm{V_i}$ es el volumen celular y $\mathrm{x_i}$ son las coordenadas del centro celular.

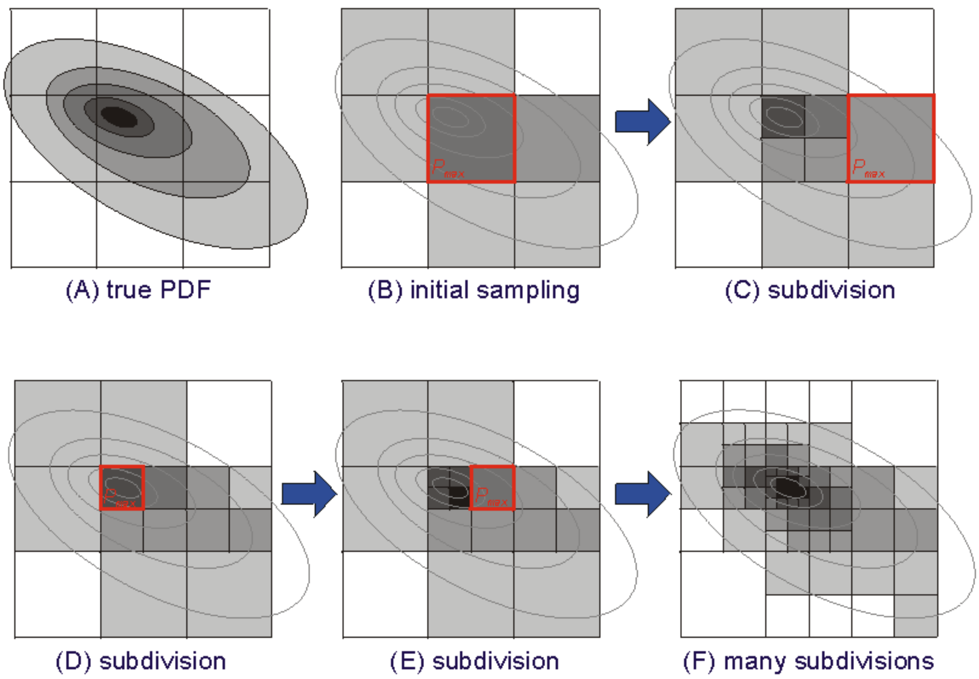
- El núcleo del método es una lista ordenada $\mathrm{L_P}$ de valores de probabilidad $\mathrm{P_i}$ para todas las celdas muestreadas previamente:



- El procedimiento de muestreo de oct-tree se inicializa mediante un muestreo global del espacio de búsqueda completo en una cuadrícula gruesa y regular (B a continuación). Se determina el valor de desajuste $g_i(x)$ en el centro de cada celda de la cuadrícula, se calcula la probabilidad P_i y la celda se inserta en la lista de probabilidades L_P en la posición correspondiente a su probabilidad P_i.

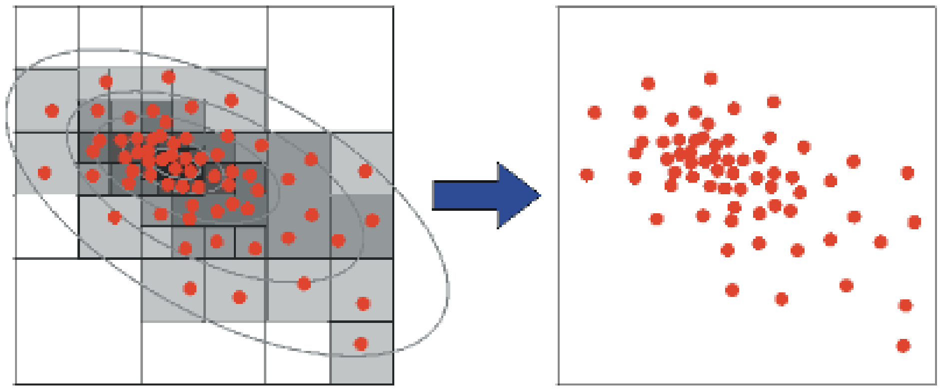
- A continuación, se repiten los siguientes pasos (CE a continuación) hasta que se haya alcanzado un número predeterminado de evaluaciones del problema directo u otro criterio de terminación:

  - 1. La celda C_{max} con la mayor probabilidad P_{max}(cuadrados rojos a continuación) se obtiene de la lista ordenada L_P.
  - 2. C_{max} se divide en 8 células secundarias.
  - 3. El desajuste y la probabilidad P_i se calculan para cada una de las 8 celdas secundarias.
  - 4. Las 8 nuevas celdas se insertan en la lista ordenada L_P de acuerdo con su P_i.
  - 5. Repetir.



- Este procedimiento recursivo converge rápidamente, produciendo una estructura de celdas oct-tree de celdas que especifican valores PDF de ubicación en el espacio 3D (F arriba). Esta estructura de oct-tree tendrá un mayor número de celdas en las regiones de PDF más alto (desajuste más bajo) y, por lo tanto, proporciona un muestreo de importancia aproximada del PDF (A arriba).

- Finalmente, las muestras en 3D extraídas de la estructura del oct-tree (abajo) dan una representación útil y compacta del PDF .



El siguiente ejemplo se ilustra mediante proyecciones en 2D de tales muestras en 3D.

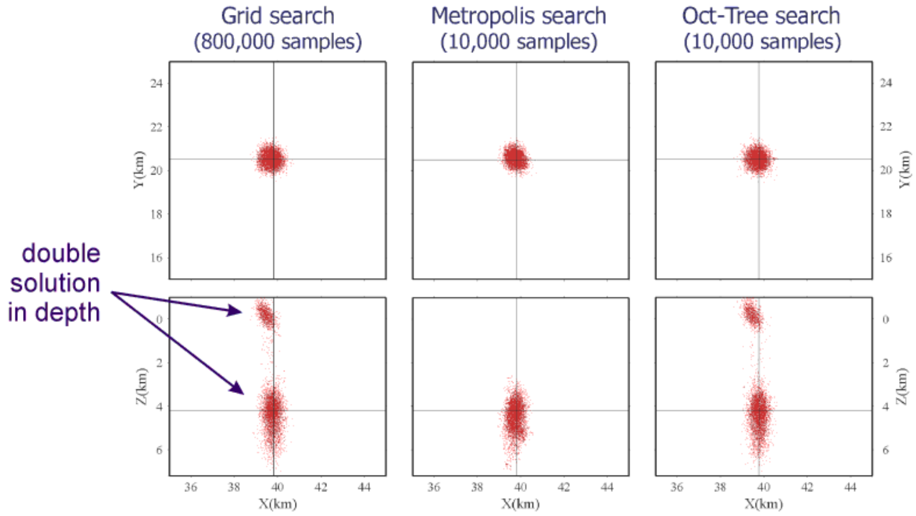
Ejemplo: ubicación de un terremoto con una solución doble.

- Una exhaustiva búsqueda en la cuadrícula (abajo, izquierda) muestra el PDF completo de la ubicación; Este PDF muestra dos regiones distintas de alta probabilidad a diferentes profundidades.

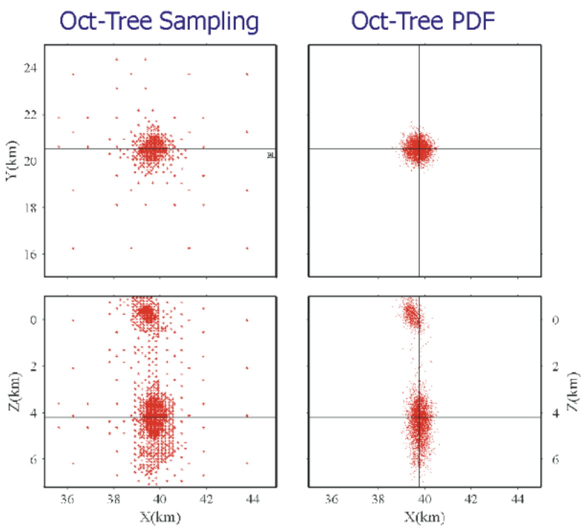
- El método oct-tree (derecha) identifica ambos volúmenes de solución y, por lo tanto, es más completo que un enfoque de recocido simulado en Metrópolis (centro), que identifica solo la solución más profunda.

- Ambos métodos son aproximadamente 100 veces más rápidos que la búsqueda de cuadrícula, pero solo el método oct-tree produce una imagen de la solución PDFeso es casi idéntico al de la exhaustiva búsqueda de cuadrícula.

- Los métodos de recocido simulados por Oct-tree y Metropolis son solo 10 veces más lentos que los algoritmos de ubicación linealizados estándar, que son difíciles o imposibles de aplicar con modelos 3D.



- Una imagen de todos los centros celulares visitados por el método oct-tree (golpe, izquierda) muestra que este método muestrea globalmente mientras produce un muestreo de importancia eficiente: es decir, la distribución de los centros celulares sigue de cerca la distribución de muestras del PDF final (derecha).



9. Código

```
#include <bits/stdc++.h>
#include "CImg.h"
using namespace std;
using namespace cimg_library;

string int_to_bin(int n){
    bitset<8> bin_x(n);
    return bin_x.to_string();
}

int bin_to_int(int n){
    string bin_string = to_string(n);
    int number = stoi(bin_string , 0, 2);
  return number;
}


class Node{
    public:
        int RED,GREEN,BLUE,contador ,nivel;
        Node *pSon[8];
    public:
        Node(int r=0,int g=0, int b=0){
            RED = r;
            GREEN = g;
            BLUE = b;
            contador = 1;
            nivel = 0;
            pSon[0]=pSon[1]=pSon[2]=pSon[3]=pSon[4]=pSon[5]=pSon
    [6]=pSon[7]=NULL;
        }
};
class Octree{
    public:
        Node *root;
        int Num_color;
        vector<Node *> pallete;
        map<int ,vector<Node *>> nivel_node;
    public:
        Octree(){
            root = new Node (0,0,0);
            Num_color =0 ;
        };

        void insert(Node *&temp,vector<int> positions ,int r,int
    g, int b){
```

```
            Node * aux  = root;
            while(!positions.empty()){
                int i = positions.front();
                if (!aux->pSon[i]){
                    aux->pSon[i] = new Node (r,g,b);
                    Num_color+=1;
                }
                else{
                    aux->pSon[i]->RED+=r;
                    aux->pSon[i]->GREEN+=g;
                    aux->pSon[i]->BLUE+=b;
                    aux->pSon[i]->contador+=1;
                }
                positions.erase(positions.begin());
                aux = aux->pSon[i];
            }
        }

        void inserttt(Node *&temp,vector<int> &positions,int r, int g, int b){
            if(positions.empty()) return;
            int elemt= positions.front();
            if(temp->pSon[elemt]==0){
                temp->pSon[elemt] = new Node (r,g,b);this->Num_color+=1;
            }
            else{
                temp->pSon[elemt]->RED+=r;
                temp->pSon[elemt]->GREEN+=g;
                temp->pSon[elemt]->BLUE+=b;
                temp->pSon[elemt]->contador+=1;
            }

            positions.erase(positions.begin());
            return insert(temp->pSon[elemt],positions,r,g,b);
        }

        void promedio_color(Node *&temp){
            if(!temp){
                return;
            }
            else{
                for (int i=0;i<8;i++){
                    Node *aux = temp->pSon[i];
                    if(aux){
                        temp->RED       +=aux->RED;
                        temp->GREEN     +=aux->GREEN;
                        temp->BLUE      +=aux->BLUE;
                        temp->contador  +=aux->contador;
```

```
                        Num_color-=1;
                    }
                    temp->pSon[i]=0;
                }
            }
        }

        vector<int> index(int r,int g,int b){
            string R = int_to_bin(r);
            string G = int_to_bin(g);
            string B = int_to_bin(b);
            vector<int> salida;
            for(int i=0;i<8;i++){
                string elemeto_aux;
                elemeto_aux.push_back(R[i]); elemeto_aux.push_back(G[i]); elemeto_aux.push_back(B[i]);
                salida.push_back(bin_to_int(stoi(elemeto_aux)));
            }
            insert(root,salida,r,g,b);
            return salida;
        }
        void print(Node *tmp){
            if (tmp == NULL) return;
          queue<Node *> q;
          q.push(tmp);
          while (q.empty() == false){
            int nodeCount = q.size();
            while (nodeCount > 0){
              Node *node = q.front();
              cout<<"("<<node->RED<<") ";
              q.pop();
              if (node->pSon[0]!=NULL)q.push(node->pSon[0]);
              if (node->pSon[1]!=NULL)q.push(node->pSon[1]);
              if (node->pSon[2]!=NULL)q.push(node->pSon[2]);
              if (node->pSon[3]!=NULL)q.push(node->pSon[3]);
              if (node->pSon[4]!=NULL)q.push(node->pSon[4]);
              if (node->pSon[5]!=NULL)q.push(node->pSon[5]);
              if (node->pSon[6]!=NULL)q.push(node->pSon[6]);
              if (node->pSon[7]!=NULL)q.push(node->pSon[7]);
              nodeCount--;
            }
            cout << endl;
          }
        }

        void Quantizer_Color(int numero, Node *&tmp){
          if (tmp == NULL) return;
          queue<Node *> q;
          stack<Node *> arr;
```

```
            q.push(tmp);
            int nvl=0;
            map<int,vector<Node *>> nivel_nodes;
            while (q.empty() == false){
              int nodeCount = q.size();
              while (nodeCount > 0){
                Node *node = q.front();
                node -> nivel = nvl;
                nivel_nodes[nvl].push_back(node);
                q.pop();
                      arr.push(node);
                if (node->pSon[0]!=NULL)q.push(node->pSon[0]);
                if (node->pSon[1]!=NULL)q.push(node->pSon[1]);
                if (node->pSon[2]!=NULL)q.push(node->pSon[2]);
                if (node->pSon[3]!=NULL)q.push(node->pSon[3]);
                if (node->pSon[4]!=NULL)q.push(node->pSon[4]);
                if (node->pSon[5]!=NULL)q.push(node->pSon[5]);
                if (node->pSon[6]!=NULL)q.push(node->pSon[6]);
                if (node->pSon[7]!=NULL)q.push(node->pSon[7]);
                nodeCount--;
              }
              nvl+=1;
            }

            for(int i=7;i!=0;i--){
                cout<<i<<"\t"<<nivel_nodes[i].size()<<endl;
                for(auto temp:nivel_nodes[i]){
                      int hojas = nivel_nodes[3].size()+nivel_nodes[4].size();
                    if(Num_color<=numero){cout<<nivel_nodes[1].size()<<" "<<nivel_nodes[2].size()<<endl;break;}
                      for (int i=0;i<8;i++){
                          Node *aux = temp->pSon[i];
                          if(aux){
                              temp->RED       +=aux->RED;
                              temp->GREEN     +=aux->GREEN;
                              temp->BLUE      +=aux->BLUE;
                              temp->contador  +=aux->contador;
                              Num_color       -=1;
                          }
                          temp->pSon[i]    =NULL;
                      }
                      nivel_nodes[i].pop_back();
                  }
            }
          }

          void generate_Pallete(Node *tmp){
              if(!tmp)
```

```
                return;
            else
                for(int i=0;i<8;i++){
                    if(tmp->pSon[i]){
                        pallete.push_back(tmp->pSon[i]);
                        generate_Pallete(tmp->pSon[i]);
                    }
                }
        }

        void Create_palette(int numero,vector<Node *>arr){
            numero = sqrt(numero);
            CImg<int> theImage(numero,numero,1,3,1);
            int valor = 0;
            int data = numero;
            while(!arr.empty()){
                for (int i=valor; i<data; i++){if(arr.empty()) break;
                    for (int j=valor; j<data; j++){
                        if(arr.empty()) break;
                        theImage(i,j,0,0) = arr.front()->RED/arr.front()->contador;
                        theImage(i,j,0,1) = arr.front()->GREEN/arr.front()->contador;
                        theImage(i,j,0,2) = arr.front()->BLUE/arr.front()->contador;
                        arr.erase(arr.begin());
                   }
                }
            }
            theImage.display();
        }

        //TEST
         bool sonEmpty(Node *aux){
            for (int i=0;i<8;i++){
                if(aux->pSon[i]!=0){
                    return false;
                }
            }
            return true;
        }

        tuple<int,int,int> Reduccion_Image(vector<int> &positions, Node *temp){
            Node * aux  = root;
            while(!positions.empty()){
                int i = positions.front();
                if(sonEmpty(aux->pSon[i])==true){
```

```
                        return make_tuple(aux->pSon[i]->RED/aux->pSon[i]->contador,
                        aux->pSon[i]->GREEN/aux->pSon[i]->contador,
                        aux->pSon[i]->BLUE/aux->pSon[i]->contador);
                    }
                    positions.erase(positions.begin());
                    aux = aux->pSon[i];
                }
            }


        vector<int> DesIndex(int r,int g,int b){
            string R = int_to_bin(r);
            string G = int_to_bin(g);
            string B = int_to_bin(b);
            vector<int> salida;
            for(int i=0;i<8;i++){
                string elemeto_aux;
                elemeto_aux.push_back(R[i]); elemeto_aux.push_back(G[i]); elemeto_aux.push_back(B[i]);
                salida.push_back(bin_to_int(stoi(elemeto_aux)));
            }
            auto it = Reduccion_Image(salida,root);
            salida.clear();
            salida[0]=get<0>(it);
            salida[1]=get<1>(it);
            salida[2]=get<2>(it);
            return salida;
        }
};

int main()
{
    Octree Raiz;
    string name_file;
    cout<<"Inserte nombre de imagen mas extencion: ";
    cin>>name_file;
    CImg<int> file(name_file.c_str());
    CImg<int> theImage(file.width(),file.height(),1,3,1);
    for(int i=0;i<file.width();i++){
        for(int j=0;j<file.height();j++){
            int r = file(i,j,0,0);
            int g = file(i,j,0,1);
            int b = file(i,j,0,2);
            //theImage(i,j,0,0)=r;
            //theImage(i,j,0,1)=g;
            //theImage(i,j,0,2)=b;
            vector<int> aa=Raiz.index(r,g,b);
        }
```

```
    }
    file.display();
    int entrada;
    cout<<"Numero de colores: ";
    cin >> entrada;

    Raiz.Quantizer_Color(entrada,Raiz.root);
    Raiz.generate_Pallete(Raiz.root);

    for(int i=0;i<file.width();i++){
        for(int j=0;j<file.height();j++){
            int r = file(i,j,0,0);
            int g = file(i,j,0,1);
            int b = file(i,j,0,2);
            vector<int> aa=Raiz.DesIndex(r,g,b);
            theImage(i,j,0,0)=aa[0];
            theImage(i,j,0,1)=aa[1];
            theImage(i,j,0,2)=aa[2];
        }
    }
    theImage.display();
    theImage.save_bmp("output.bmp");
    Raiz.Create_palette(entrada,Raiz.pallete);
  return 0;
}
```

test.cpp

10. Pruebas

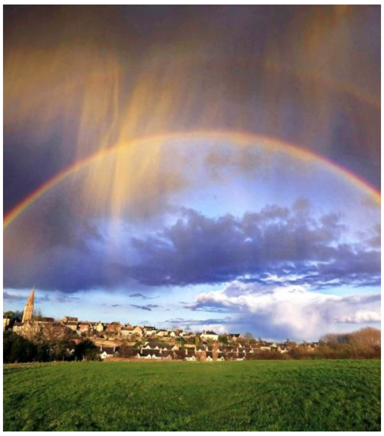


Figura 6: Imagen Original

Figura 7: Imagen reducida de 256 colores

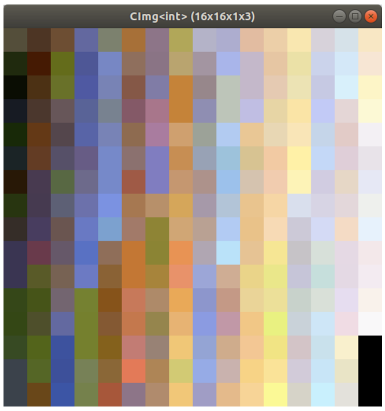


Figura 8: Paleta de 256 colores

Figura 9: Imagen reducida de 64 colores

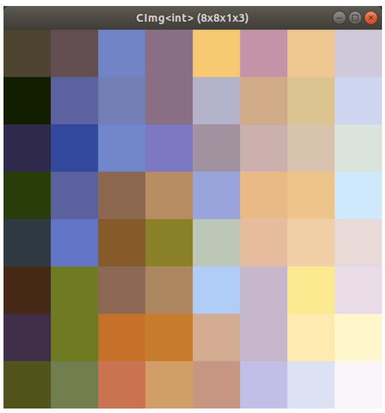


Figura 10: Paleta de 64 colores

11. Conclusiones

- permite el uso de la geometría simple de la división oct-tree de celdas rectangulares: el volumen de cada celda siempre se conoce y se puede determinar fácilmente qué celda contiene un punto dado. El método oct-tree debería ser aplicable en 4D, permitiendo una búsqueda sobre el tiempo de origen. Pero en problemas de dimensiones superiores, la determinación del volumen de una celda y si una celda contiene o no un punto dado puede ser difícil o imposible.
- El enfoque oct-tree se puede aplicar a la ubicación telesísmica en una tierra esférica
- El método Árbol octal es un buen método, por que al construir un objeto se puede subdividir únicamente la parte donde no se tenga la precisión adecuada, mientras que por ejemplo en la enumeración espacial es necesario dividir toda la cuadrícula, aunque en el método de enumeración espacial un objeto se puede obtener más fácilmente desde un arreglo tridimensional, sin necesidad de tener que hacer un análisis complejo.
- El método Árbol octal al combinarse con la Geometría Sólida Constructiva es muy útil para crear objetos complejos, únicamente con objetos sencillos, y ofrece buena aproximación; aunque entre más complejo sea el objeto más tiempo se tardará en crear.

12. Bibliografía

- https://hera.ugr.es/tesisugr/17693895.pdf
- Conversion and Integrationof Boundary Representations with Octrees* - T. K. Chan, 1. Gargantini
- Teoría de los OCTREES

Referencias

[1] Dan S. Bloomberg. Color quantization using octrees. *Leptonica.*