# Universidad Nacional de San Agustín

**Facultad de Producción y Servicios**

**Escuela Profesional de Ciencia de la Computación**


**Curso :**

**COMPUTACIÓN GRÁFICA**

**Tema :**

**Informe Scanner 3D**

**Docente :**

Juan Carlos Gutiérrez Cáceres

**Alumnos:**
Christian ,Condori Mamani
Israel Santiago, Pancca Mamani

**AREQUIPA - PERÚ**

**2017**

# Índice

# 1. Introducción

El escáner 3D es un dispositivo que analiza objetos o entornos para poder obtener los datos de sus formas y su apariencia. Los datos obtenidos son utilizados para la reconstrucción digital del objeto, crea un modelo tridimensional que puede ser utiizado en diversas aplicaciones: industriales, entretenimiento, videojuegos, ingenieria inversa, control de calidad, arqueología, entre otros.

El obejtivo del escáner 3D es crear una nube de puntos de mnuestras geométricas sobre la superficie del objeto. Mediante estos puntos puede extrapolarse la fomra del objeto mediante un proceso de reconstrucción digital, además de obtener los colores del objeto si se tuvieron en cuenta en la implementación.

# 2. Construcción básica de un escáner 3D

Para la construcción básica de un escáner 3D está conformada principalmente por un sensor, un emisor de señales y un sistema de procesamiento de datos. Los dispositivos láser son los más utilizados para emitir la señal que se va a capturar y procesar, siendo los láseres de línea los más populares. Se utilizan cámaras como sensores gracias a que son muy eficientes para la captura en datos. Por último, el sistema de procesamiento de datos es un software diseñado para transformar y procesar las señales provenientes de la cámara, las cuales vienen en forma de punto de nubes de puntos.

# 3. Planteamiento

## 3.1. Requisitos necesarios para armar el escenario

Para el escenario se necesito los siguientes instrumentos:

- **La cámara de un celular**: Basta con una de baja resolución, ya que con la segmentación se espera que la línea del láser sea lo suficientemente clara para realizar las operaciones necesarias para hallar las coordenadas del objeto geométrico.
- **Un puntero láser de color rojo**:Usaremos un láser que proyecte una línea recta y delgada. El choque de la luz del láser con el objeto nos dará una idea de la profundidad de los puntos contenidos en la línea proyectada. Si no se tuviera un láser de línea; y por el contrario contamos con un láser de punto; es posible generar la línea proyectado la luz desde el lado cóncavo de la base con el LED.
- **Plataforma de una caja músical para el giro**: El objeto geométrico a rotar será colocado en esta base que girará a una velocidad adecuada para tomar los puntos alrededor de los 360 grados de vistas del objeto.
- **Un LED**: Será utilizado para generar una línea que nos ayudará a segmentar.

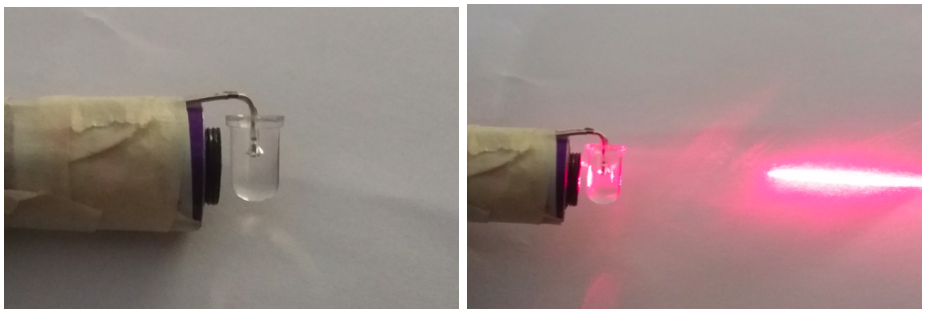

Figura 1: Láser pegado al LED.

La cámara de celular lo utilizaremos como una cámara web cam gracias a una aplicación llamado IP Webcam, esta aplicación lo podemos encontrar en PlayStore, la finalidad de este programa es poder acceder a la cámara bajo una dirección IP generada por la aplicación en nuestro caso nos genero **http://192.168.42.129:8080**

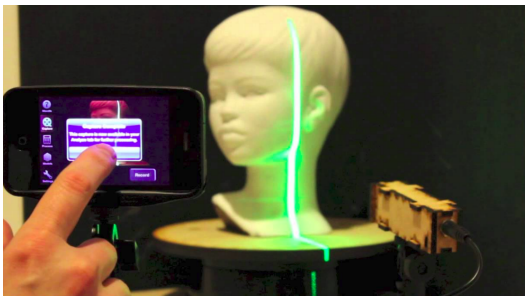

Figura 2: Uso de IP Webcam.

## 3.2. Fijando variables

Para comenzar, requerimos frames suficientes, que nos den la información de 360 grados de vistas alrededor del objeto. Calibramos la velocidad de rotación de la base y alineamos la cámara mirando de frente al centro del objeto a escanear.

El láser es colocado a un ángulo $\theta$ de la cámara, de tal manera que su dirección apunte al centro del cuerpo geométrico, Para definir este ángulo, se observan dos casos. Si el $\theta$ es lo suficientemente grande, podría hacernos perder información del objeto, en el caso de que el objeto tenga superficies cóncavas; por otro lado, si $\theta$ es demasiado pequeño, la curva generada en el objeto se observaría plana debido a la cercanía con la cámara. Esto se observa en las figuras 3 y 4. Así, se trata de elegir un $\theta$ adecuado. Por ahora consideraremos adecuado un ángulo de 45 grados.

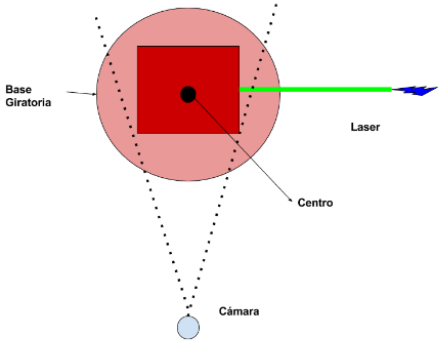


Figura 3: Ángulo Theta demasiado grande.

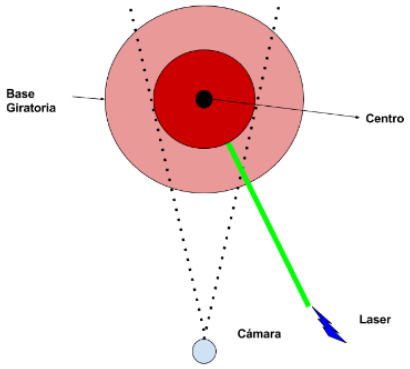


Figura 4: Ángulo Theta demasiado pequeño.

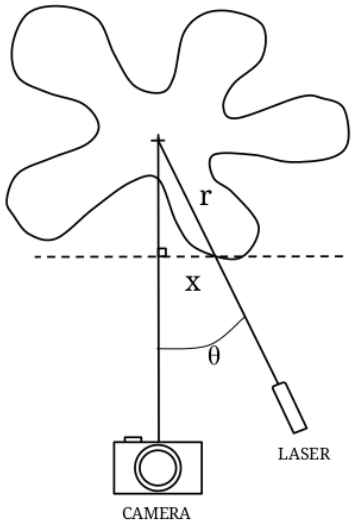


Figura 5: Esquema del espacio y variables a utilizar.

# 4. Segmentación

Para extraer la distancia x del frame a tratar. Primero es necesario hacer una segmentación para ignorar todos los elementos que no sean útiles. En este caso solo re- querimos la línea proyectada por el láser, ya que partimos asumiendo que el centro del frame coincide con el centro del objeto (eje y).

Utilizaremos la librería OpenCV para capturar los frames. Esta librería nos provee varias funciones útiles para ejecutar la segmentación.

- **TheVideoCapturer**:Detecta por defecto la cámara que tenemos en nuetra computadora para una mejor comodidad de la cámara utilizaremos la cámara de un celular es por ello que le indicamos la dirección URL que nos generó el IP Webscam
- **TheVideoCapturer.read(imgOriginal)** Extrae cada frame y lo almacena en **imgOriginal** que es un tipo de Mat, la estructura que OpenCV utiliza para almacenar una matriz.

- **cvtColor**: Convierte una imagen de un espacio de color a otro debido a que la imagen generada es RGB y nosotros necesitaremos una imagen de escala de gris, se creo una variable Mat **imgHSV** que será la que contenga nuestra imágen de escala de gris.

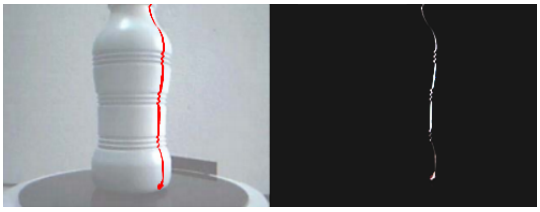

Figura 6: Segmentación.

- **inRange**: Previamente definimos valores mínimos y máximos para cada variable en el formato HSV. Estas variables serán útiles para captar solo el color que nos interesa; en este caso, la luz del láser. Deberemos calibrar esto cada vez por medio de trackBars. Al elegir un rango de colores. El resultado será una iamgen binaria en la que solo es visible (en blanco) la luz del láser.
- **erode**: La función erosiona la imagen de origen utilizando el elemento de estructuración especificado que determina la forma de un vecindario de píxeles sobre el cual se toma el mínimo.
- **Dilate**: La función dilata la imagen de origen utilizando el elemento de estructuración especificado que determina la forma de un vecindario de píxeles sobre el cual se toma el máximo.
- **Canny**: Nos permite filtrar los bordes del objeto que vamos a escanear gracias a erode y dilate que nos muestra los bordes de un tamaño considerable para poder captarlos.
- **findContours**: Una vez filtrado el ruido con canny pasaremos a la detección de los contornos y los guardaremos en $vector < vector < Point >> contours$

# 5. Calculando distancias

En la figura 5 se muestra la estructura de la que disponemos para el cálculo. Teniendo como base el frame segmentado. Tomaremos cada punto en Y y aproximaremos la coordenada en X de acuerdo a la distancia x mostrada en el esquema. Esta distancia será obtenida en píxeles. Se pretende escalar esta distancia con la equivalencia:

$$100px = 1,0$$

en caso que la cámara sea de 480px de ancho. Este depende y se ajsuta de acuerdo a la resolución de la cámara. La coordenada Y toma la misma escala. Para conseguir la coordenada $Z$, que representa la profundidad, de acuerdo al esquema es necesario hallar la distancia $r$, desde el punto actual al eje y que coincide con el centro del cuerpo. Usando trigonometría, y teniendo en cuenta que conocemos x, luego de la aproximación visual, y $\theta$ de acuerdo a la disposición entre la cámara y el láser.
Aplicamos la siguiente fórmula:

$$r.\sin(\theta) = x$$

Una vez obtenida la distancia $r$, solo queda calcular la coordenada $Z$. Con Pitágoras simple tenemos:

$$r^2 = x^2 + Z^2$$

Ya tenemos el punto $Z$ para un $X$ y $Y$ actuales. Por cada ángulo recorreremos toda la línea del láser tanto en $X$ y $Y$ .

Para completar todos los puntos alrededor de los 360 grados, será necesario aplicar rotación de puntos en 2D. Ya que la coordenada Y no se verá modificada. X y Z serán modificadas de acuerdo al ángulo $\alpha$ que irá de 1 a 360 grados. Luego:

$$x' = x.cos(rotacion)$$
$$z' = x.sin(rotacion)$$

De esta manera, obtenemos las coordenadas $(X, Y, Z)$ para completar nuestra nube de puntos del objeto geométrico.

# 6. Generando el archivo .obj

Finalmente, procedemos a escribir las coordenadas para cada punto en un archivo **obj.** Un archivo obj. tiene la estructura mostrada en el siguiente ejemplo.

# Lista de vertices geométricos, con $(x, y, z[, w])$ coordenadas, $w$ es opcional y automaticamente se calcula con la función findContours que guardará todos los puntos en una matriz de puntos, los puntos son números enteros donde la posición $y$ se mantiene en el pixel ubicado y $x$ y $z$ son nuevos puntos generados por la rotación de puntos 2D, como ejemplo tenemos:
v −150 529 17
v ...
...

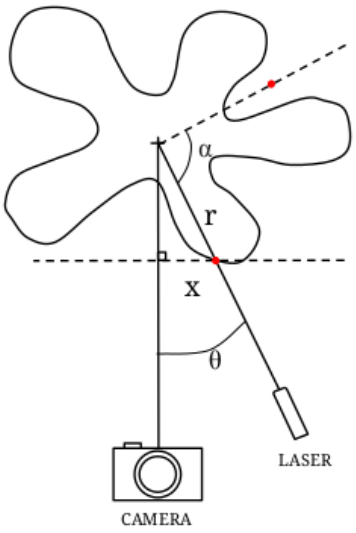


Figura 7: Nuevo punto luego de la rotación.

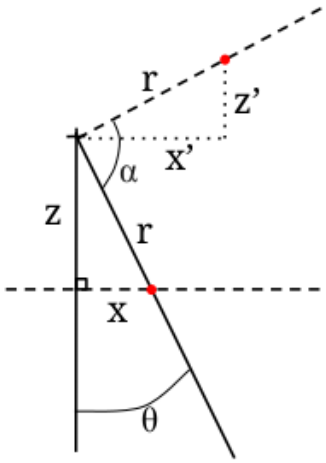


Figura 8: Nuevo punto luego de la rotación.

# 7. Explicación del código

## 7.1. main

Nuestro main principal llama a la función **recog(()** que tiene el objetivo de analizar los bordes y generar un archivo de salida obj.

```
1 int main(int argc, char *argv[]) {
2     recog();
3     return 0;
4 }
```

## 7.2. Función void recog()

En la línea 2 usamos la función **TheVideoCapturer** que abre a la cámara que tiene la dirección URL.
En la línea 3 hace una consulta si no ha sido abierto la cámara nos muestra el error y sale de la función.
Sí se reconoce la cámara ejecutaremos apartir de la línea 7.
La línea 15 verifica si no se puede leer el frame para mandar un mensaje.
La línea 18 crea una imagen que contendra en modo escala de gris.
La línea 19 que tiene la función cvtColor transforma dicha imagen en escala de gris y lo guarda en **imgHSV**.
La línea 20 crea una variable que contendrá los contorno del objeto apuntado por el láser.

```
1  void recog(){
2      TheVideoCapturer.open("http://192.168.42.129:8080/video?x.mjpeg");
3      if( !TheVideoCapturer.isOpened()){
4          cout<<"no se pudo abrir la camara"<<endl;
5          return;
6      }
7      int cont = 0;
8      int x,y;
9      char key = 0;
10     float ang = 1.3;
11     while(key != 27){
12             Mat imgOriginal;
13             bool bSuccess = TheVideoCapturer.read(imgOriginal);
14             if(!bSuccess){
15                 cout<<"No puede leer frame de el video stream"<<endl;
16                 continue;
```

```
            }
            Mat imgHSV;//imagen esala de gris
            cvtColor(imgOriginal, imgHSV, COLOR_BGR2HSV);//transforma la imagen
                de RGB a imagen gris en imgHSV
            Mat imgThresholded;//crea otra variable tipo Mat
            calibration(imgOriginal); // linea de referencia al centro
            inRange(imgHSV, Scalar(iLowH, iLowS, iLowV),Scalar(iHighH,iHighS,
                iHighV), imgThresholded); //colores en rango que vamos leer
            //utiliza imgHSV y los escalares son los minimos y maximos,
                imgThresholded es la salida
            erode(imgThresholded,imgThresholded,Mat()); //ensancha la linia
                reconocida
            dilate(imgThresholded,imgThresholded,Mat());//hace mas preciso la
                linia

            Mat salida =imgThresholded.clone();//saca una copia
            Canny(imgThresholded, salida, 100, 200);//<ojo>toma las variables de
                 los bordes
            vector< vector< Point> > contours; //los puntos que usaremos
            map<int,int> mPoints;

            findContours(salida, contours, RETR_LIST,CV_CHAIN_APPROX_SIMPLE);//
                toma los puntos de salida y los guarda en contours
            getMediaPoints(contours,mPoints);//saca la media de los contornos
                externos

            vector<Point> mp;
            int _X,_Z;
            //rotcion
            cout<<cont<<"n "<< rotation<<"   ang:  "<<ang<<endl;
            for(const auto &myPair : mPoints){ // iteracion, mas rapido?
                    if(myPair.first < 790){
                        triangulation(myPair.second,myPair.first,_X,_Z);//
                            Convierte los puntos 2D en 3D
                        vector<int> tmp;
                        tmp.push_back(_X);
                        tmp.push_back(myPair.first + 50);
                        tmp.push_back(_Z);
                        pointCloud.push_back(tmp); // malla de puntos
                    }
                }
                rotation+= 1.33*((22/7)/180.0);
                ang= ang + 1.3;//para que llegue a 360
               // if(key == 115)
                saveObj();//recibe la malla y lo guarda

            if(ang > 365) break;   //355
            resize(salida, salida, Size(salida.cols/2, salida.rows/2)); //
                tamanio dela pantalla a la mitad
            imshow("canny",salida);//muestra la ventana
            cont++;
            key = 0;
            key=waitKey(20);
    }
}
```

## 7.3. void saveObj()

Esta función tiene el objetivo de crear un archivo de salida de tipo **.obj** donde será visualizado por el programa MeshLab.

```
void saveObj(){
    ofstream myfile("oobjetoFFs.obj");
    string ver = "v ";
    for(int i=0;i<pointCloud.size();++i){
       myfile << "v "<<pointCloud[i][0]<<" "<<pointCloud[i][1]<<" "<<pointCloud[
           i][2]<<endl;
    }
    myfile.close();
```

```
}
```

## 7.4. void getMediaPoints(...)

Esta función nos ayuda a poder sacar la media de los contornos del láser debido a que cuando el láser choca con el objeto da un borde grueso y al ver en escala de gris este se muestra en 2 línias entonces frente a este problema usamos esta función que tiene el objetivo de sacar la media de los contornos del las 2 línias del contorno.

```
void getMediaPoints(vector<vector<Point> > contours, map<int,int> &mPoints){
    if(contours.size()){
        map<int,int>::iterator it;
        for(int i=0; i<contours.size(); ++i){
            for(int j =0; j < contours[i].size(); ++j){

                it = mPoints.find( contours[i][j].y );
                if( it != mPoints.end() ){
                    mPoints[contours[i][j].y ] = ( mPoints[ contours[i][j].y ] +
                        contours[i][j].x )/2;
                }else{
                    mPoints[ contours[i][j].y ] = contours[i][j].x;
                }
            }
        }
    }
}
```

## 7.5. void triangulation(...)

Convierte los puntos 2D en 3D aplicando rotación en los puntos $\_X$ y $\_Z$

```
void triangulation(int x, int y, int &_X, int &_Z){
    x = (240-x)*0.4; //480//es el tamanio de la pantalla en 2
    _X = x*cos(rotation);
    _Z = -x*sin(rotation);
}
```

## 7.6. void calibration(...)

Muestra una línea de referencia en el centro de la pantalla, esto nos servirá para poder ubicar al objeto en el centro de la pantalla.

```
void calibration(Mat &imgOriginal){
    CvPoint mid_bottom, mid_top;
            mid_bottom.x = 640/2;
            mid_bottom.y = 0;
            mid_top.x = 640/2;
            mid_top.y = 480;
            IplImage frame_ipl = imgOriginal;

     cvLine(&frame_ipl, mid_bottom, mid_top, CV_RGB(0,255,0), 2);
}
```

## 8. Ejecución del código

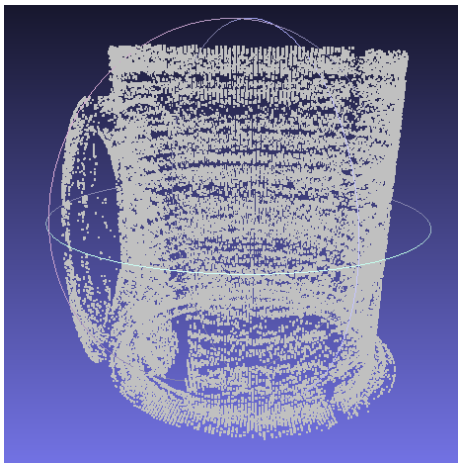

Figura 9: Nube de puntos del objeto.

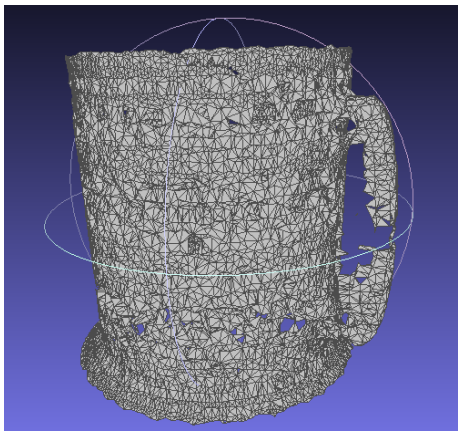

Figura 10: Objeto texturizado.

## 9. Conclusiones

- En un caso ideal, cada frame corresponde a un ángulo. Pero no será así necesariamente. Es importante calibrar la velocidad de la base giratoria antes de empezar a calcular los puntos.
- De acuerdo al ángulo $\theta$ elegido, se perderán datos para zonas cóncavas. Esto podría solucionarse usando dos lásers. El ángulo entre ámbos láser sería 90 grados para tratar de salvar estas zonas de pérdida.
- En este caso solo presentamos la nube de puntos en el archivo OBJ. Sin embargo, es posible completar la triangulación de las faces para mostrar un modelo 3D con más información.

## 10. Bibliografía


- `http://repositorio.uac.edu.co/bitstream/handle/11619/1368/TMCT%200011C.pdf?sequence=2`
- `https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html`