**ESE 358 Computer Vision**

**ECE, SBU, M. Subbarao**

**Project 2. Image Formation: Computer Animation and Augmented Reality**

**(Revised with help from TA C. Orlassino)**

**Image formation in a pin-hole camera: Mapping from 3D object coordinates to 2D image coordinates under 3D rigid-body motion.**

**Note: See pages 33-60 in the Szeliski’s text book, 2021 edition.**

**BACKGROUND: Image and video formation in a pin-hole camera: the forward and the inverse processes**

This project makes explicit the process of the **forward image** **and video formation process** in a pin-hole camera. The output of this project forms the input to many vision algorithms that try to **invert the image formation process** to recover the original description of the 3D scene and motion from 2D images and videos.

1. **Digital representation of a 3D object** (e.g. cube) approximated by a set of polygons (e.g. squares). The geometric information is represented by a set of polygons and the photometric information is represented as texture-maps on the polygons. The polygons themselves are represented as a sequence of edges Ek=(Vik,Vjk) for k=1 to n, with n sides. For each polygon, a set of closely spaced points on a grid lying on that polygon are computed, and the texture-map values at each of these points are stored. For simplicity, a cube with square faces and texture-map on a single face are used in the project. Therefore, hidden point elimination will not be considered. It can be added a brute-force method where the projection of all points onto the image plane are computed and stored first, and then only the points closest to the camera along each direction of view is determined and used for display.
2. **Motion parameters:** Initial position, initial translation velocity, uniform acceleration, axis of rotation, and uniform rotational velocity, are read as input parameters.
3. **Geometric transformation and perspective projection:** For time t=0 seconds to t=maxtime seconds at intervels of dt seconds, compute the image points, and display them on the screen. This step involves moving and rotating the 3D points by applying the rotation and translation transforms, and projecting the resulting 3D points onto the image plane, and projecting the texture-map, and displaying the image.
4. **Forward process: 3D Scene to image:** The forward image/video formation process in this project makes explicit the 3D digital representation objects with their geometric (polygonal faces) and photometric (texture-map) components, input data (3D scene with possibly multiple objects), parameters of geometric transformation and motion (R and T matrices), perspective-projection matrix (K) involving the camera parameters, and the role of texture-maps.
5. **The output of this forward process is the video sequence.** This video sequence forms the input to the 3D vision algorithms. One desired output of such vision algorithms is the recovery of the original 3D scene description (that was used as input in generating the video sequence) from the given video sequence as input such as the 3D object description (3D shape and texture-map), and motion parameters (translational velocity and acceleration, rotational velocity, etc).
6. **Inverting image formation process, Calibration, tracking/matching, and features:** The vision algorithms will need to solve the problem of inverting the forward image/video formation process. This involves camera calibration to determine the matrix K (perspective projection matrix), tracking/matching corresponding points in the texture-maps in the video sequence to determine the shape (using stereo) and motion parameters (R, T, acceleration, etc.).
7. The problem of tracking/matching pixels could be solved efficiently by computing image features and matching/tracking the image features (e.g. edges, corners, sift vectors).
8. After solving the matching/tracking problem, a set of typically non-linear equations are solved to obtain the 3D parameters of the scene. This may involve regularization to account for noise and ill-conditioned/under-constrained cases.
9. Computation of image features is also useful in classification problems for identifying objects in images (e.g. dogs or cats).
10. This project can be extended from cubes to tetrahedrons, prisms, spheres, face models, and more complicated 3D scenes, including multiple objects with different motion parameters, panoramic image and video generation for AR/VR etc.

**PART 1: COMPUTER ANIMATION:**

**Goal: Animate a rotating and translating cube with a texture-map on one face of the cube.**

**On Blackboard, you are given a program template with some parts of the code removed. Fill in the missing code so that the program performs the tasks specified below.**

**Overview:**

In the template you are given the following values:

* **V1 to V8**: The 8 vertices of a 3D cube in millimeters specified in a World Coordinate System (WCS), where each edge is of **length** mm:

**V6<--------- length -------->V8**

**/ | / |**

**/ | / |**

**/ | / |**

**V2--|--------------------------V3 |**

**| | | |**

**| | | |**

**| | | |**

**| V7----------------------- |---V5**

**| / | /**

**| / | /**

**| / | /**

**V1---------------------------- V4**

* **T0**: The origin of the WCS in the camera coordinate system (CCS)
* **Theta0**: Initial rotation of the WCS (zero for this project)
* **W0**: angular velocity in degrees/second
* **F**: focal length of camera
* **Velocity**: the x, y, z translational velocity of the cube
* **Acc**: the x, y, z translational acceleration of the cube
* **P**: pixel size in mm
* **Rows, cols**: size of image in pixels

Handling translation of the WCS:

The translation of the WCS wrt the CCS with time t in seconds is given by

**T**=**T0**+ **velocity**\*t + 0.5 \* **acc** \* t \* t;

* **T0**, **Velocity**, and acceleration **acc** are vectors given in the CCS.

Handling rotation of the WCS:

The WCS is rotated by an angle **theta** around a given axis along the unit vector **N0** passing through the origin of the WCS.

The rotation angle **theta** of the WCS wrt the CCS along the same axis **N0** with time **t** in seconds is given by:

**theta** = **theta0** + **w0**\*t

* The above equation specifies angle theta as a function of time t with initial angle **theta0** added to rotational angle **w0**\*t where **w0** is the uniform rotational/angular velocity.
* **theta0** and **w0** are given

To rotate, an axis of rotation **N0** must be determined:

* The axis of rotation **N0** in this project is specified to be the vector passing through **V1** and **V8** (i.e., **N0** is a unit vector parallel to **V8**-**V1**) at time t=0.0 in the CCS and remains the same for the rest of the time period. For now, ignore other possibilities.
* **N0** is used to construct the 3x3 matrix **N** (see eq. 2.32 in the attached pages from Szeliski’ book).
* **N** is used to compute the 3x3 rotational matrix **R** (see eq. 2.34 in the attached pages from Szeliski’ book). This **R** changes as **theta** changes with time.

Handling texturing:

One face of the cube that will be textured is the face bounded by V1, V2, V3, V4.

For faster output results, use a smaller image (e.g. the 50x50 image einstein50x50v.jpg) as the texture image. After your program works on the small image, try it on a bigger image (e.g. the 132x132 image einstein132.jpg) for better looking results. You can try with other images that you wish.



50x50 zoomed; 132x132

For each pixel in this image, the program finds the 3d location of this pixel on the front face of the cube. These WCS points are then mapped to CCS (and then sampled over the pixel space) to obtain the final image.

One of the task that you need to do is this: replace the call to the function cv2.line() to draw line between two pixels with your own code, by not calling any other library function. You can do this with the method below:

You can compute all the pixels lying on the image line connecting two image points **q1**(x1,y1) and **q2**(x2,y2) as follows (note: (x1,y1) and (x2,y2) are row and column indices of pixels **q1** and **q2** ). A vector equation for the line is

**q = q1 +** c **\* u** for 0.0 <= c <= d

where **u** is a unit vector along **q2-q1**, and c is a scalar number in the range 0 to d where d is the length of the vector **q2-q1**, in units of pixels. In order to avoid gaps, c can be incremented in units of 0.5 pixels in the range 0.0 to rounded value of d to the nearest integer.

For c= 0.0 to d in steps of 0.5, compute

**q = q1 +** c **\* u**

set image\_gray-level value of pixel located at **q** to be 255

#note: background pixels have been initialized to 0

End

**Optional:** A thickness parameter can be added to the line as follows. Suppose you want a line thickness of 3 pixels. If **u**=(a,b), then the two vectors normal to **u** on either side are given by **u1**= +/- (b,-a) (verify that their dot product with **u** is zero.). Then draw lines from **q1+u1** to **q2+u1**, and another from **q1-u1** to **q2-u1.**

The method above has been extended to a 3D plane to compute the (X,Y,Z) coordinates of the given texture map pixels texmap(i,j,.) in the program template.

**Skip this:** Notes on texture map resolution: The computational step of texture-mapping of an image frame at each time instant can be optimized as follows. Instead of scanning over all possible pixels at which the texture map is given (e.g. size 50x50 or 132x132), we use the estimated pixel resolution at the distance Z of each sample. Suppose the given texture map is of size NxN pixels for a square object surface patch of size LxL mm, at distance Z and normal to the optical axis, and p is the pixel size in mm and f is the focal length in mm, then, according to the perspective projection relations, it can be shown that we should roughly have (p\*N)/f = L/Z. If N>>(f\*L)/(Z\*p), then we have oversampling (some computation is wasted). If N<<(f\*L)/(Z\*p), then the given texturemap is under-sampled (and we need to do interpolation such as bilinear interpolation to assign texture map values to all the necessary pixels in the image frame). A good situation is that N is roughly 20% more than (f\*L)/(Z\*p). In the data given for the project, at time t=0, we have f=40 mm, L=10 mm, Z=500 mm, and p=0.01 mm. These numbers give N=80. Therefore the size of the given texture map should be 80x80 or more. Otherwise, there will be gaps in the computed texture maps in the image frames. Further, if the object is moving with time, then Z changes, and if the object is rotating and the surface normal of the surface patch of the object makes an angle of **phi** with respect to the optical axis, then we need N>=(f\*L\*cosine(phi))/(Z\*p), at each time instant.

**Part 1 tasks:**

* **Note:** the location of each task is clearly specified in the comments of the template

1. Understand the Map2Da, MapIndex, and DrawLine functions in the template. Fill in the missing piece of Map2Da which concerns the y-value for the returned 2D point.
2. Replace the call to the function cv2.line() to draw line between two pixels with your own code. See the method above.
3. Compute the axis of rotation N0, which is u81: the unit vector corresponding to the vector (V8-V1). Use this to construct N, the matrix defined in eq 2.32 necessary to later construct the rotation matrix R (eq. 2.34).
4. Initialize the intrinsic matrix K given focal length f:
5. Compute u21 and u41, the unit vectors corresponding to the vectors (V2-V1) and (V4-V1) respectively. These are used to discover each unique 3D point on the face of the cube which will be textured.
6. When computing all 3D points on the face which will be textured, complete the code which saves the Y and Z values of these points. The X value is already computed in the template as an example.
7. Compute the rotation matrix R as defined in eq 2.34.
8. Map all eight 3D vertices to their 2D counterpart using Map2Da. Furthermore, convert from mm to pixels using MapIndex.
9. Use DrawLine to draw the 12 edges of the cube. Note that the edges join the following:

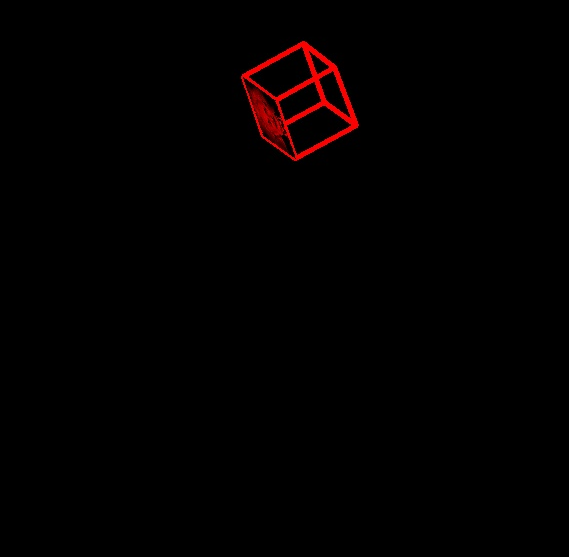
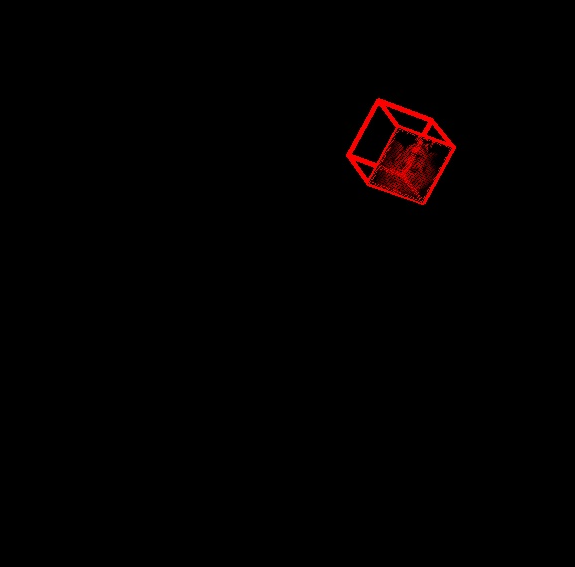
(v1,v2), (v2,v3), (v3,v4), (v4,v1),

(v7,v6), (v6,v8), (v8,v5), (v5,v7),

(v1,v7), (v2,v6), (v3,v8), (v4,v5).

1. Compute ir and jr, which are the row and column indices of the output image which corresponds to the texture map color at (i, j).

Example output for part 1 at frame 8 out of 24, and frame 16 out of 24:

**PART 2: AUGMENTED REALITY:**

**Goal: Augment a given real-world image of a 3D scene (e.g. live broadcast of a sporting event) by introducing the cube object generated by your program in the foreground of the scene. In addition, generate a 3D stereo-video of the augmented scene by introducing a second camera. The original red cube animation is combined with blue cube animation corresponding to the second (stereo) camera. If the resulting video is seen after wearing a red and blue colored eye-glasses, the cube will be perceived in 3D (as in 3D movies).**

* Note that the code template does not show you where to perform part 2 tasks

**Task 1) Add a background:**

Superimpose the image sequence generated in your previous section with a given background image. All pixels of the background image that are not occluded by the wire-frame cube must retain their original values in the 3D scene, and all pixels that are occluded by the cube must have the pixel values generated from the cube.

* For a clear view of cubes, set the 600x600 background image to be a gray-level image.

**SKIP THIS PART: Task 2) AR for Stereo Vision**

In addition to the camera in the previous section, a binocular stereo camera is created by placing a second camera. All of the axes in the CCS of the second camera is perfectly aligned with the corresponding axes of the first camera with the only difference being that the origin of the CCS of the second camera is located at (10, 0, 0) mm. Note that 10 mm is the baseline distance along the X-axis of the first camera.

Compute the animation sequence with the output of the first camera shown in red color, and the output of the second camera shown in blue color. The output of both cameras must be shown on the same single output color image frame. When this color image frame is viewed with a Red-Blue eye-glasses on the left and right eyes respectively, it would produce the perception of a 3D cube translating and rotating in 3D space.

Submit the following:

* The Python source code
* The background image you used
* Screenshots of the animation
  + Provide at least four, well-spaced apart screenshots of the animation.

