iBubbleBurst Game

1- Summary

The goal of this game is to burst all the bubbles that come towards you with your head- pose. You just move your head left-right and up-down to position the cross-hair and burst the bubbles.

Each bubble burst worth 100 points, and if you miss one, you lose 150 points. Some graphical measures guide you throughout the game. The levels are automatically change based on score. In new levels, the bubbles speed becomes more furious.

2- Key Algorithms

The program has been written with C++ in OSX X-Code with the help of OpenCV, OpenGL, and SDL libraries. The key algorithm here is the head- pose estimation based on raytracing and fitting a solid virtual cylinder in position of player’s head. This algorithm has two major components:

i- Fitting the Virtual Cylinder

Fitting a virtual cylinder in position of player’s head is easy with using HAAR classifiers to find the bounding box of the head on OpenCV’s captured frame. After the binding-box is found, the height and width of the cylinder is estimated. The z-axis position of the cylinder can be arbitrary, because any value only will change future calculation scale. We assume a value between 5 and 10 for z-axis. Now we can find the face features using Harris-corners in OpenCV, and de-project them using raytracing to find their 3D counterpart positions on the cylinder. So we have both 2D coordinates and their corresponding 3D coordinates on the cylinder.
ii- **Tracking the Head and Updating Cylinder position**

Since we already have both 2D and 3D positions of the features, we can use KLT Optical Flow algorithm to track then in next frame. At this point, we have 2D and 3D positions on Frame “i” and 2D positions on frame “i+1.” With these information, we can use perspective-n-points algorithm to estimate the rotation and translation of the virtual cylinder, and update our object.

Apart from Head-Pose Estimation (HPE) algorithms, some other key algorithms that have been used in the game are as follows:

i- **Plane-Line Intersection**

This algorithm has been used in two places: updating the cross-hair position and checking if the bubble can be burst by the cross-hair.

To show the cross-hair to player, an imaginary plane is positioned in front of player. The Head-Pose vector is intersected with this plane, and the intersection point is the cross-hair position.

To check if the bubble can be burst by the cross-hair, first the bubble plane is checked for intersection with a line from origin to cross-hair to the bubble plane. If the intersection is found, then the boundaries are checked to see if the point is inside the circle.

ii- **Random Content Generation**

Bubbles are randomly generated based on their z-axis position. All of them have the same radius, but their colors change to yellow and then red as they become closer to the player. Once they are missed by the user, or player has successfully burst one, a new one will be randomly created again.
iii- **Visual Effects**

If a successful hit is detected, 4 lines will be drawn from corners to the cross-hair for visual effects. Also, the background changes to dark gray momentarily. If player misses a bubble, the background will be red momentarily.

iv- **Automatic Level Upgrade and Speed Change**

When the score goes above 1,000, the second level begins with a little higher pace. At 3,000, Level 3 starts. At 5,000, Level 4 starts. I have never reached 10,000, but there is a Level 5.

v- **Synchronized Sounds-FX**

Sound effects are synchronized with the game with the help of SDL library frequency generator. Successful hit is generated from 180Hz and missed are set at 500Hz.

3- **Limiting Factors**

The calibration phase requires the player to be full frontal to the camera. The lighting condition also changes the performance of the tracking algorithms.

4- **Design Perspective**

The original idea was to design a Bug Zapper game with head movements, but after changing the bugs to circles, it could very well be a bubble bursting game without killing off any insects!