Bubble Blaster VR

In this tutorial, we'll be making a simple bubble blaster for the HTC Vive.

As always, start by making a new Unity project. I built this project using Unity 5.3.4, but any of the 5.4+ versions should work. This walkthrough covers:

- Basic environment building
- Creating our bubble and generating new bubbles
- Popping the bubbles and keeping score
- Teleporting around using the SteamVR SDK
- Saving the best time locally to compete with friends

After we start our project, the first thing that we're going to want to do is build a small environment for our application. I'm a huge fan of Dreamscapes, but you can use any of the terrain assets you want. If you need a primer on Unity terrains, I wrote up a pretty in-depth tutorial for the editor – while it's for an older version of Unity, quite a few of the basics still apply!

For this particular application, I wanted a demo that I could have people play in a relatively short amount of time: think a 3-5 minute experience. I decided to keep the terrain relatively small, and set the resolution at 50x50.

\

You also should add a small sign, with a Canvas and Text element attached to the sign with the tag 'Sign' on Text object, to show the score and timer, which we'll add later.

Bubbles

Once you've created your scene, the next step will be to add in a bubble. We'll be using this as a prefabricated object that we can clone to generate new bubbles, but our first step will be making our own!

I made my own bubble by using one of the Unity built-in material shader types, but you can use another GameObject if desired:

1. Create a sphere GameObject
2. Create a new material with the following properties:
    a. Shader: Standard
    b. Rendering Mode: Transparent
    c. Metallic: 0.192
    d. Smoothness: 0.812
    e. Albedo: I created a basic light blue 512 x 512 .png texture in Gimp
3. Assign the material to the sphere
4. Set the Bubble to a new layer called "Bubbles"

I'm also not going for a terribly realistic experience, so I decided to create a rainbow firework effect instead of a regular "pop" effect. It was one part "I can do what I want, this is VR!" and one part "I don't really want to animate liquid soap physics". To do this, I added two particle systems as children to my bubble – be creative here! Or use my particle effects, either one works. I'm not trying to tell you how to make your game. For the rainbow sparkle effect that I chose, I used two variations on these component settings:

Particle System

Duration	1.00
Looping / Prewarm / 3D Start Rotation / Play on Awake	False
Start Delay / Start Rotation / Randomize Rotation Direction	0
Start Lifetime	0.5
Start Speed	5
Start Size	Random between two constants: 0 and 1
Start Color	White
Gravity Modifier	2
Simulation Space / Scaling Mode	Local
Max Particles	1000

Emission / Shape

Rate	10 / Time
Shape	Sphere

Radius	0.5
Emit from Shell / Random Direction	False

Over Lifetime (Limit Velocity, Color, Size)

Limit Velocity: Separate Axes	False
Limit Velocity: Speed	3
Limit Velocity: Dampen	1
Color over Lifetime	Random between two curves – rainbow gradients (one Red-Yellow, one Green-Blue)
Size over Lifetime	Curve – downward slope

For the renderer, I used the defaults with a simple particle from a free package from the asset store.

To make it so that our bubbles "pop" when shot, we need to add in a behavior script to our bubble object. In the hierarchy, click the sphere for your Bubble and under the inspector, click 'Add Script'. Create a new C# script called "BubbleScript". Before moving into the next step, double check that your particle systems are children of the bubble!

At the top of the BubbleScript class, add the following line:

```
    GameController _controller;
```

In the Start method, set the GameController instance to the following – don't worry at this point if you see an error.

```
void Start () {
     _controller =
GameObject.FindObjectOfType<Terrain>().GetComponent<GameController>();}
```

Finally, add the following method under the `Update()` method:

```
    public void Pop()
    {
       GetComponent<MeshRenderer>().enabled = false;
       foreach (ParticleSystem _system in
       gameObject.GetComponentsInChildren<ParticleSystem>(true))
       {
           _system.Play();
       }

       _controller.SendMessage("NewBubble");
       Destroy(gameObject);
    }
```

BEHIND THE CODE:

IN OUR POP METHOD, THE FIRST THING THAT WE DO IS GET THE MESH RENDERER COMPONENT OF THE BUBBLE – THIS IS WHAT SHOWS US THE BUBBLE – AND HIDE IT. WE THEN LOOP THROUGH EACH OF OUR PARTICLE SYSTEMS (I USED TWO, BUT YOU COULD ADD AS MANY HERE AS YOU WANTED) USING A FOR-EACH LOOP, WHICH SAYS "FOR EACH OF THE ITEMS IN THIS

LIST, DO SOMETHING TO THAT ITEM." IN OUR CASE, WE GET A PARTICLE SYSTEM (REPRESENTED BY THE _SYSTEM VARIABLE) AND THEN PLAY IT. FINALLY, WE SEND OUR GAME CONTROLLER A MESSAGE TO CREATE THE NEW BUBBLE.

Once you've completed this, save your bubble as a prefab to use in the scripts.

Game Controller

Back over in Unity, the next step is to add a "master controller" that will handle the game play components. I decided to attach this to the terrain, which stays static, but you could really put it on anything that wasn't being destroyed. Create another C# script – I called mine "GameController". This will be the guts of our program, so we'll break it down step by step.

The first thing that you'll want to do is put in our global variables for our GameController. These are going to cover a few things:

Type	**Variable**	**Description**
`int MIN` `int MAX`	`-15` `15`	The bounds for our bubbles to spawn
`int SCORE`	`0`	The score to track the bubbles popped
`float TIMER;`	`0.0f`	Track how long it takes to collect the bubbles
`bool PLAYING`	`true`	Whether or not the game is active or has already been won
`public GameObject originalBubble;`	`<Assign in inspector>`	The bubble to clone

The code should look like this, placed directly above the Start method:

```
int MIN = -15;
int MAX = 15;

int SCORE = 0;
float TIMER = 0.0f;

bool PLAYING = true;

public GameObject originalBubble;
```

Make sure to set your originalBubble to be the bubble prefab you created!

BEHIND THE CODE:

WE STORE THREE OF OUR VARIABLES AS INTEGERS (INT) WHICH ARE WHOLE NUMBERS. OUR TIMER IS MORE PRECISE, SO WE'LL USE A FLOATING POINT NUMBER (FLOAT). WHETHER OR NOT THE PERSON IS PLAYING IS A TRUE OR FALSE STATEMENT, SO WE USE A BOOL FOR THAT. LASTLY, WE MAKE OUR ORIGINAL BUBBLE GAME OBJECT PUBLIC, SO WE CAN SEE AND ASSIGN THE VALUE IN THE INSPECTOR LATER ON.

For our game loop, we're going to want to:

a) Check if the game has been won yet – PLAYING is set to true
b) Update our timer to track how long the player has been playing

c) Update our sign to show the latest timer and score

To begin, we're going to add a few more methods to our GameController class. The first one that we're going to add is a helper method to format our timer nicely – we want to separate out the seconds and minutes, then trim our timer objects so that it is easy to read on the sign.

Our `FormatTimer()` method will return a string with the time elapsed in a nicely packaged mm:ss format:

```
// Helper method convert the timer to minute/second format
string FormatTimer()
{
    if(TIMER <= 60.0f)
    {
        return (Mathf.Round(TIMER *100) / 100).ToString();
    }
    else
    {
        int displayTimeMin = (int)(TIMER / 60.0f);
        float displayTimeSec = Mathf.Round((TIMER % 60) * 10 / 10);
        if(displayTimeSec < 10)
        {
            return displayTimeMin.ToString() + ":0" + displayTimeSec.ToString();
        }
        else
        {
            return displayTimeMin.ToString() + ":" + displayTimeSec.ToString();
        }
    }
}
```

BEHIND THE CODE:

THE FIRST THING THAT WE DO IS CHECK IF MORE THAN 60 SECONDS HAVE ELAPSED. IF NOT, WE RETURN A STRING THAT ROUNDS OFF THE SECONDS TO TWO DECIMAL POINTS. IF WE HAVE BEEN PLAYING FOR MORE THAN 60 SECONDS, WE GET THE MINUTES BY DIVIDING OUR TIME IN SECONDS (TIMER) BY 60. WE USE THE MODULO OPERATOR ('REMAINDER') TO GET THE NUMBER OF SECONDS. IF WE ARE BETWEEN 0-9 SECONDS, ADD AN ADDITIONAL '0' IN FRONT FOR CONSISTENT FORMATTING. WE RETURN OUR FORMATTED STRING IN EACH CASE.

The next function that we're going to add is a really straightforward, single-line function to call in our game loop that uses the function we just wrote to display our formatted timer on our sign object, which we tagged earlier:

```
// Display the time
void DisplayTime()
{
    GameObject.FindGameObjectWithTag("Sign").GetComponent<Text>().text = "Time: "
                                                        + FormatTimer()
                                                        + System.Environment.NewLine
                                                        + "Bubbles: " + SCORE + "/20";
}
```

BEHIND THE CODE:

WE FIRST USE GAMEOBJECT.FINDGAMEOBJECTWITHTAG AND SPECIFY OUR SIGN, WHICH WE ASSIGNED IN THE INSPECTOR EARLIER. THIS TAKES OUR TEXT OBJECT, FINDS THE COMPONENT FOR THE TEXT ITSELF (IT'S A LITTLE CONFUSING – THERE IS A COMPONENT TEXT WITH A CAPITAL T AND AN ATTRIBUTE TEXT WITH THE LOWERCASE T) AND THEN ASSIGNS IT A STRING. WE CALL OUR FORMATTIMER() METHOD TO ADD IT TO OUR STRING, AND INCLUDE THE NUMBER OF BUBBLES (OUT OF 20) THAT THE PLAYER HAS FOUND.

\

We have two more functions to add to our game controller, after which we'll go ahead and update our main loop! The first of these two is a public function called "NewBubble". This will be called from our Bubble Script, which we'll also update, to tell the game controller that a bubble has been popped. This function:

- Is public, because we need to send it a message from the BubbleScript
- Increases the player score
- Generates a new bubble within the bounds of the board
- Adds the bubble to the board

The code:

```
// Generate a new bubble
    public void NewBubble()
    {
        SCORE++;

        float xVal = Random.Range(MIN, MAX);
        float yVal = Random.Range(2.0f, 5.0f);
        float zVal = Random.Range(MIN, MAX);

        Vector3 bubblePos = new Vector3(xVal, yVal, zVal);
        Instantiate(originalBubble, bubblePos, originalBubble.transform.rotation);
    }
```

BEHIND THE CODE:

WE UPDATE OUR SCORE VALUE FIRST, THEN CREATE AN X,Y, AND Z VALUE FOR OUR NEW BUBBLE'S LOCATION. OUR X AND Z COORDINATES SPECIFY WHERE ON THE BOARD BETWEEN THE MINIMUM AND MAXIMUM VALUES THE BUBBLE WILL BE, AND THE Y VALUE IS SET TO BE SOMEWHERE AT OR SLIGHTLY ABOVE EYE LEVEL.

The final function to add is an end-game state, which will stop the timer and let the player know how they did. We will use Unity's Player Preferences to store the best time, and update that if the current time beats the saved time:

```
// End game, save score
    void EndGame()
    {
        PLAYING = false;
        if (!PlayerPrefs.HasKey("BestTime") || PlayerPrefs.GetFloat("BestTime") < TIMER)
        {
            PlayerPrefs.SetFloat("BestTime", TIMER);
            GameObject.FindGameObjectWithTag("Sign").GetComponent<Text>().text = "Time: "
                                                    + FormatTimer()
                                                    + System.Environment.NewLine
                                                    + "New High Score!";
        }
    }
```

BEHIND THE CODE:

WHEN THE GAME ENDS, WE FIRST SET PLAYING TO FALSE, THEN CHECK IF THERE IS A BEST TIME SAVED OR IF WE'VE BEATEN AN EXISTING BEST TIME. IF SO, WE SET THE BEST TIME, AND UPDATE THE SIGN TO INDICATE A NEW HIGH SCORE HAS BEEN REACHED.

Finally, to finish off our game controller script, we're going to add the final lines of code into our `Update()` method:

```
    // Update is called once per frame
    void Update () {
     if(PLAYING)
     {
         TIMER += Time.deltaTime;
         DisplayTime();

         if (SCORE >= 20)
         {
             EndGame();
         }
     }
}
```

BEHIND THE CODE:

OUR UPDATE METHOD IS VERY SIMPLE, BECAUSE OF THE HELPER METHODS WE WROTE EARLIER. WE CHECK IF THE PLAYER IS STILL PLAYING, AND IF SO, WE UPDATE THE TIMER, DISPLAY IT, CHECK THE SCORE, AND END THE GAME IF THE PLAYER HAS GOTTEN 20 POINTS.

With that, our game controller code is finished! We'll now be able to track all of the functionality of our bubble blaster with this script.

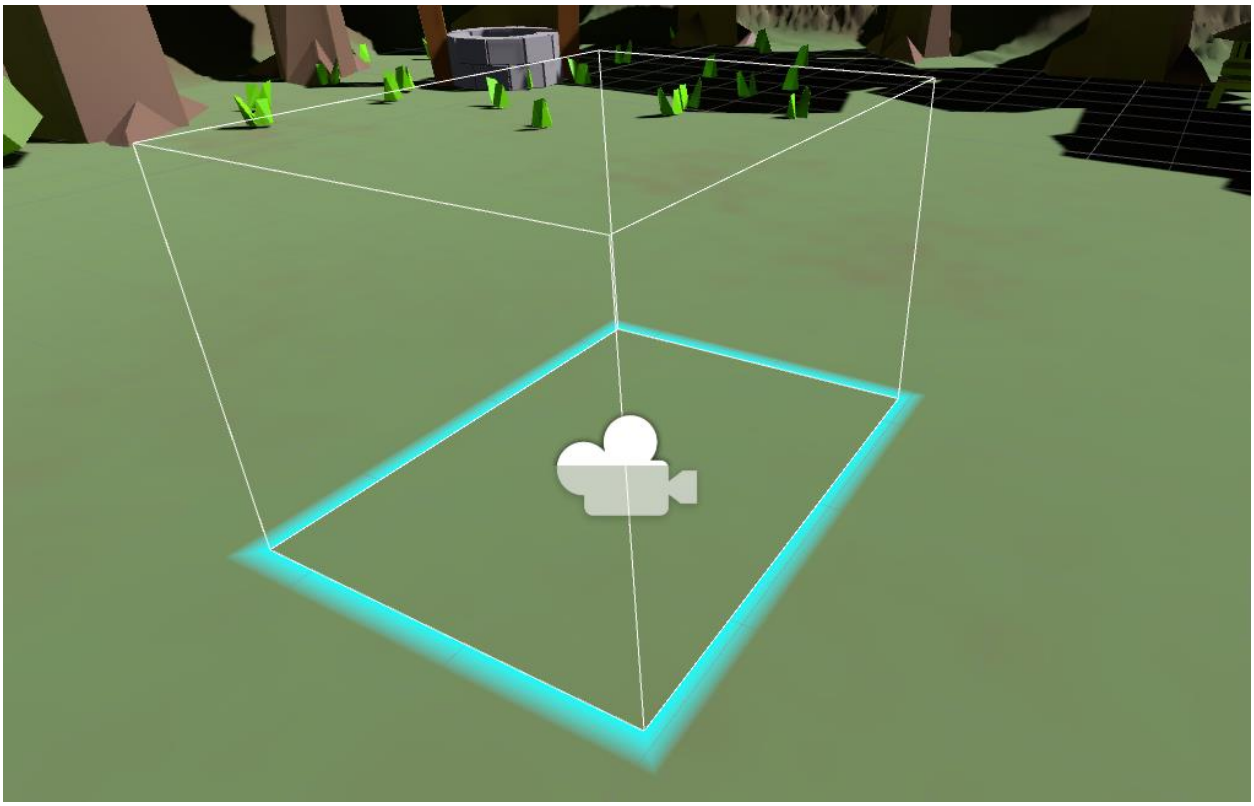
Vive Support

Now for the fun part – adding in Vive support!

First, we're going to want to bring in the star of the show – the HTC Vive support! From the Asset Store, download and import the SteamVR plugin. This contains a whole bunch of delightful assets, including prefabs and extra functionality scripts that make developing for the Vive work really nicely.

1. Delete your main camera from the scene
2. From the SteamVR package, open the Prefabs folder and drag the CameraRig object into your scene.

You should see an outline of the Vive lighthouse system show up around your camera, and you'll want your camera to be aligned with the floor of your scene.



At this point, I highly recommend trying on the headset and running your app to see how everything looks. Standing inside of a world of your own making is an incredible experience! You'll also get a chance to evaluate scale of your environment, as well as see how Vive's Chaperone looks in your app.

You'll notice that the controllers are automatically tracked at this point, but don't actually do anything when the buttons are clicked. Let's fix that!

Adding Teleportation

One of the things we'll probably want to do is add in a teleportation option so that we're able to move around our terrain freely, even outside of the bounds of the Chaperone box. Valve has made this super easy with the SteamVR plugin, and contains a teleport script for us to use in the Extras folder. Find the

SteamVR_Teleporter.cs script – this file contains everything we'll need for now to teleport around our room!

Under the CameraRig prefab in the Hierarchy, you'll find two game objects that represent the controllers. Since I'm right-handed, I'm going to use that one to shoot the bubbles, so I went with the Left Controller to teleport.

1. Drag and Drop the SteamVR_Teleporter.cs script directly onto one of the Controller objects
2. Select where you'd like the teleporter to go. For me, I chose the Teleport Type Use Terrain, but fair warning: this does mean you can teleport a lot of random places!
3. Check the "Teleport On Click" option and you're all set!

Adding a Laser Pointer

To make it easier to see where we're pointing to shoot down our bubbles, we're going to use the included laser pointer script on our shooting controller. For me, I picked the right controller for this task.

1. Drag and drop the SteamVR_LaserPointer.cs script directly onto the other Controller object
2. Choose the color of your laser pointer

Super simple!

Creating a Player Script

Now that we've added in the basics of the SteamVR plugin, we're going to add a player controller to our controller that shoots the bubbles as they're generated. This is super easy with the SteamVR plugin!

Create a new script called PlayerScript.cs and add it to the controller you'll be using to shoot from. At the top of the file, add the following variables:

```
BubbleScript _activebubble;
SteamVR_TrackedController controller;
```

In the Start() function, include the following lines of code:

```
// Use this for initialization
void Start () {
    controller = GetComponent<SteamVR_TrackedController>();
    if (controller == null)
    {
        controller = gameObject.AddComponent<SteamVR_TrackedController>();
    }
    controller.TriggerClicked += new ClickedEventHandler(Fire);

}
```

BEHIND THE CODE:

WE ARE FIRST CHECKING TO SEE IF THERE IS ALREADY A STEAMVR_TRACKEDCONTROLLER COMPONENT TO OUR OBJECT. IF THERE IS, WE JUST SET IT, BUT IF THERE ISN'T, WE WILL CREATE THE COMPONENT PROGRAMMATICALLY SO THAT WE CAN INTERACT WITH IT. LASTLY, WE ACCESS THE "TRIGGER CLICKED" PROPERTY OF THE CONTROLLER, WHICH IS WHAT STORES ACTIONS THAT ARE PERFORMED WHEN THE TRIGGER IS PULLED, AND ADD A NEW EVENT HANDLER THAT CALLS THE METHOD "FIRE" WHEN THE TRIGGER IS PULLED.

You don't have to add anything into the Update function, but we will be creating a new method called Fire() that will serve as our clicked event handler for the controller:

```
// Fire when trigger on controller clicks
 void Fire(object sender, ClickedEventArgs e)
 {
     Debug.Log("Fired");
     int layerMask = 1 << 8;
     RaycastHit _hit;
     if (Physics.Raycast(transform.position, transform.forward * 10,
                         out _hit, 10.0f, layerMask))
     {
         _activebubble = _hit.collider.gameObject.GetComponent<BubbleScript>();
         _activebubble.SendMessage("Pop");
     }
 }
```

BEHIND THE CODE:

WE HAVE CREATED A SPECIAL TYPE OF METHOD CALLED AN EVENT HANDLER, WHICH IS CALLED SPECIFICALLY WHEN A GIVEN EVENT OCCURS – IN THIS CASE, THE TRIGGER PULL. WE ARE TELLING OUR RAYCAST TO IGNORE ALL LAYERS BUT THE BUBBLES BY SETTING A LAYER MASK, SHOOT OUT TEN UNITS IN FRONT OF US, AND IF IT HITS A BUBBLE, TELLS THAT BUBBLE TO POP ITSELF BY USING SENDMESSAGE.

That's the whole thing! You should be able to launch your game, and shoot some bubbles while teleporting around – despite being a simple app, it's actually surprisingly fun. The basic mechanics also lend themselves to a lot of different game play elements, so enjoy!

Additional Resources

Just A/VR Show Episode covering the SteamVR portion of this tutorial:
https://channel9.msdn.com/blogs/misslivirose/Setting-up-SteamVR-in-Unity-for-the-HTC-Vive

Full code on GitHub:
https://github.com/misslivirose/vive_bubbleblaster/