# Current Trends in Gaming

## Augmented Reality – Week 1

### Context

This sheet is an accompaniment to the lecture and covers the process and code needed. It’s a reference so that you don’t need to worry about keeping pace with the project coded in real-time.

### Activity

**Step One – Create your AR Project**

Refer to the instructions linked from the canvas for your specific kind of device. There are a number of specific frameworks, tools and techniques you’ll need. It wouldn’t be a bad idea to take that empty project and copy it somewhere for a blank template of your own.

**Step Two – Setting up the AR Environment**

Once you’re all set up for your device, there are some ARFoundation objects you’ll need. In the GameObject menu you’ll find an XR submenu. Add the following game objects to your scene:

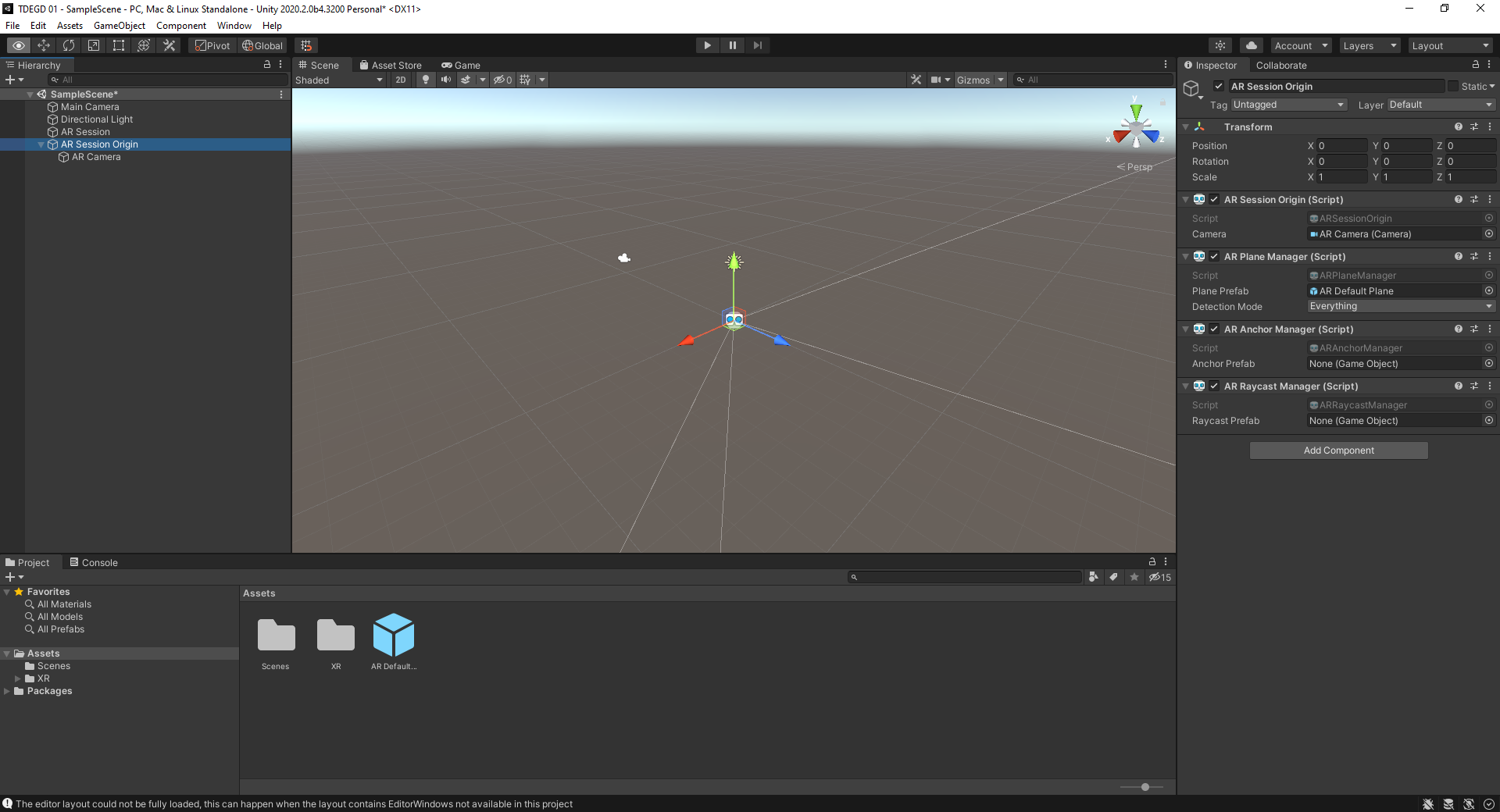
* AR Session
* AR Session Origin

On to the AR Session Origin object you’re going to add several components. Do this from the Add Component button on the inspector:

* AR Plane Manager
* AR Anchor Manager
* AR Raycast Manager

The AR Plane Manager will need to be configured with a default plane prefab. Add one of these to your scene from Game Object > XR > AR Default Plane. Drag the plane you create into the appropriate location in your AR Plane Manager, drag it into your project explorer to create a prefab, and then assign that prefab to your AR Plane Manager. Delete it from your scene once it’s setup.

Your scene should look something like this:



It would be worthwhile at this point actually trying to get it running on your device to see if everything is set up properly. So let’s do that.

**Step Three – Deploying to your device**

Again, refer to the instructions on the canvas for your specific kind of device. Essentially it’ll break down to going File > Build Settings and choosing the appropriate target. You’ll then go into ‘Player Settings’ and change some things to match your handset.

Connect your handset to your computer (again, refer to the instructions on the canvas for how this needs to be setup). Then choose ‘build and run’.

A few moments later you should see your first AR app running on your phone. If not, refer back to the instructions for whatever step you missed. If that fails, refer to your lecturer to identify whatever step **I** missed.

Your project won’t do anything except detect planes, so as long as it can do that you’re fine.

**Step Four – Adding some Content**

Next up we’re going to add in our instantiable prefab. Our little robot. Our little buddy. First of all, grab the **RobotSphere** asset from the asset store. You can use a different asset if you prefer but I can’t help debug that if it does anything weird. I’d suggest avoiding for now any asset that includes animations that include any kind of translation as part of their process because they don’t always work cleanly with AR.

Our first script is going to handle the initial spawning of the asset onto a nearby surface. Create a new script and call it RobotBehaviour. Add it to the AR Session Origin object.

Go into the tag manager and add a new tag – SpawnedObject. We’re going to add this to the robot we create in the game so that we can easily track it later.

Now, let’s put in some code for the script. Version one, that doesn’t bother itself with anything other than where the camera is facing, looks like this:

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  using UnityEngine.XR.ARFoundation;  using UnityEngine.XR.ARSubsystems;  public class RobotBehaviour: MonoBehaviour  {  public GameObject robotPrefab;  public Camera myCamera;  public float cooldown, cooldownCount;  private static ILogger logger = Debug.unityLogger;    void Start()  {  cooldown = 2;  myCamera = this.gameObject.transform.Find  ("AR Camera").gameObject.GetComponent<Camera>();  }  void Update()  {  cooldownCount += Time.deltaTime;  if (cooldownCount > cooldown && Input.touchCount == 2) {  cooldownCount = 0;  doSpawnRobot();  }    }    public void doSpawnRobot() {  GameObject robot;    robot = Instantiate (robotPrefab,  myCamera.gameObject.transform.position +  (myCamera.gameObject.transform.forward \* 1.0f),  myCamera.gameObject.transform.rotation);    robot.transform.localScale = new Vector3(3, 3, 3);  robot.tag = "SpawnedObject";  logger.Log ("spawned at " + robot.transform.position.x +  ", " + robot.transform.position.y + ", " +  robot.transform.position.z);    }    } |

You’ll need to drag the asset prefab into the appropriate location in the inspector, but once you’ve done that you’ll be able to run the game and tap with two finger son the screen to create an instance of it wherever you are currently looking. It’ll do a little animation and then enter into a dormant state.

Our script here has done several things – implement a cooldown (so that you don’t flood the screen with assets), create a reference to the AR Camera (so you know where you are looking) and scaled down the robot so it fits comfortably on the screen in front of you. This is just a simple straightforward instantiation though so you will see problems.

* It ignores everything in the environment
* It floats in the air
* You can’t do anything with it

So let’s change some of that, first by making it so that we instantiate it on one of those planes that the game is trying to hard to identify.

**Step Five – Spawning on Surfaces**

Our next step is to instantiate our robot on its nearest plane. For this we need to do a bit of raycasting, from the viewport of the camera, to the nearest identified plane. We’ll use that as the basis for where we create the robot when we tap. Our code changes quite a lot to make this happen. Bold shows new or changed code.

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  using UnityEngine.XR.ARFoundation;  using UnityEngine.XR.ARSubsystems;  public class RobotBehaviour: MonoBehaviour {  **private ARRaycastManager rays;**  public GameObject robotPrefab;  public Camera myCamera;  public float cooldown, cooldownCount;  private static ILogger logger = Debug.unityLogger;  void Start() {  cooldown = 2;  myCamera = this.gameObject.transform.Find  ("AR Camera").gameObject.GetComponent < Camera > ();  **rays = this.gameObject.GetComponent < ARRaycastManager > ();**  }  void Update() {  cooldownCount += Time.deltaTime;  if (cooldownCount > cooldown && Input.touchCount == 2) {  cooldownCount = 0;  doSpawnRobot();  }  }  public void doSpawnRobot() {  GameObject robot;  **Vector3 screenCenter;**  **bool hit;**  **ARRaycastHit nearest;**  **List <ARRaycastHit> myHits = new List <ARRaycastHit> ();**  **screenCenter = myCamera.ViewportToScreenPoint(new Vector3(0.5f, 0.5f));**  **hit = rays.Raycast(screenCenter,**  **myHits,**  **TrackableType.FeaturePoint | TrackableType.PlaneWithinPolygon);**  **logger.Log("Hit: " + hit);**  **if (hit == true) {**  **nearest = myHits[0];**  **robot = Instantiate(robotPrefab, nearest.pose.position**  **+ nearest.pose.up \* 0.1f, nearest.pose.rotation);**  robot.transform.localScale = new Vector3(3, 3, 3);  robot.tag = "SpawnedObject";  logger.Log("spawned at " + robot.transform.position.x + ", " +  robot.transform.position.y + ", " + robot.transform.position.z);  }  }  } |

Try this out and you’ll see that when you spawn a robot it will only work if a plane is detected, and if a plane **is** detected that’s where your robot goes.

**Step Six – Anchoring**

This will all work just fine, but objects created in AR have a tendency to drift as we move around the environment. As such, our final step here is going to be to **anchor** the object firmly to the plane. For that we need to make available all of the currently unused helper components we added in step two. Specifically, the AR Anchor Manager and the AR Plane Manager. We need to programmatically access their APIs to do the following:

* Check to see if the thing our raycast hit was **actually a plane**
* Handle the anchoring appropriately depending on what was hit.

So, we adjust our script accordingly:

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  using UnityEngine.XR.ARFoundation;  using UnityEngine.XR.ARSubsystems;  public class RobotBehaviour: MonoBehaviour {  private ARRaycastManager rays;  public GameObject robotPrefab;  public Camera myCamera;  public float cooldown, cooldownCount;  **private ARAnchorManager anc;**  **private ARPlaneManager plan;**  private static ILogger logger = Debug.unityLogger;  void Start() {  cooldown = 2;  myCamera =  this.gameObject.transform.Find  ("AR Camera").gameObject.GetComponent<Camera>();  rays = this.gameObject.GetComponent<ARRaycastManager>();  **anc = this.gameObject.GetComponent<ARAnchorManager>();**  **plan = this.gameObject.GetComponent<ARPlaneManager>();**  }  void Update() {  cooldownCount += Time.deltaTime;  if (cooldownCount > cooldown && Input.touchCount == 2) {  cooldownCount = 0;  doSpawnRobot();  }  }  public void doSpawnRobot() {  GameObject robot;  Vector3 screenCenter;  bool hit;  ARRaycastHit nearest;  List<ARRaycastHit> myHits = new List <ARRaycastHit>();  ARPlane plane;  ARAnchor point;  screenCenter = myCamera.ViewportToScreenPoint(new Vector3(0.5f, 0.5f));  hit = rays.Raycast(screenCenter,  myHits,  TrackableType.FeaturePoint | TrackableType.PlaneWithinPolygon);  logger.Log("Hit: " + hit);  if (hit == true) {  nearest = myHits[0];  robot = Instantiate(robotPrefab, nearest.pose.position  + nearest.pose.up \* 0.1f, nearest.pose.rotation);  robot.transform.localScale = new Vector3(3, 3, 3);  robot.tag = "SpawnedObject";  logger.Log("spawned at " + robot.transform.position.x + ", "  + robot.transform.position.y + ", " + robot.transform.position.z);  **plane = plan.GetPlane(nearest.trackableId);**  **if (plane != null) {**  **point = anc.AttachAnchor(plane, nearest.pose);**  **logger.Log("Added an anchor to a plane " + nearest);**  **} else {**  **point = anc.AddAnchor(nearest.pose);**  **logger.Log("Added another anchor " + nearest);**  **}**  **robot.transform.parent = point.transform;**  }  }  } |

You won’t see the benefits of this right away, but you’ll find your objects are a lot more stable if you get into the habit of doing a proper anchor.

**Step Seven – FUS-RO-DAH**

Our next step here will be to make our robot move when we press on the screen. For this, we need to add a collider to the robot prefab because it doesn’t automatically come with one. Add a box collider for the easiest debugging process, but a mesh collider will also be fine.

Note, when you add your collider go into the inspector to make sure it’s added with sensible proportions. I found setting the size to 0.2, 0.2, 0.2 worked fine but your mileage may vary.

Your also going to need a rigidbody attached, so you may as well do that while you’re there.

Next, we’re going to add a new function to our script – it’ll apply a force in the direction the camera is facing. Note here that it uses Raycasts rather than AR Raycasts – the former is what you use for virtual objects, the latter what you use for objects detected by the AR architecture.

|  |
| --- |
| public void doFusRoDah() {  RaycastHit[] myHits;  Ray r;    r = myCamera.ScreenPointToRay(Input.GetTouch(0).position);  myHits = Physics.RaycastAll (r);  foreach (RaycastHit hit in myHits) {  logger.Log ("Detected " + hit.transform.gameObject.name);    if (hit.transform.gameObject.tag == "SpawnedObject") {  logger.Log ("Applying force");  hit.transform.gameObject.GetComponent<Rigidbody>().AddForce  (r.direction \* 100);  }  }  } |

Our last task is to tie this into our main script, in Update, so that we can apply the appropriate function with the appropriate input.

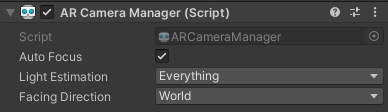
|  |
| --- |
| void Update()  {  cooldownCount += Time.deltaTime;  if (Input.touchCount == 1) {  doFusRoDah();  }  else if (cooldownCount > cooldown && Input.touchCount == 2) {  cooldownCount = 0;  doSpawnRobot();  }  } |

Run the game again, and push your robots around. Adjust the forces to taste.

**Step Eight – Light**

Our final step is to create the following script and attach it to our directional light.

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine.Rendering;  using UnityEngine.XR.ARFoundation;  using UnityEngine;  public class LightScript : MonoBehaviour  {  private Light myLight;  public ARCameraManager manager;    void Start()  {  myLight = this.GetComponent<Light>();  RenderSettings.ambientMode = AmbientMode.Skybox;  }  void OnEnable()  {  manager.frameReceived += handleChange;  }  void OnDisable()  {  manager.frameReceived -= handleChange;  }    void handleChange(ARCameraFrameEventArgs args)  {  if (args.lightEstimation.averageBrightness.HasValue) {  myLight.intensity = args.lightEstimation.averageBrightness.Value;  }  if (args.lightEstimation.averageColorTemperature.HasValue) {  myLight.colorTemperature = args.lightEstimation.averageColorTemperature.Value;  }  if (args.lightEstimation.colorCorrection.HasValue) {  myLight.color = args.lightEstimation.colorCorrection.Value;  }  if (args.lightEstimation.mainLightDirection.HasValue) {  myLight.transform.rotation =  Quaternion.LookRotation(args.lightEstimation.mainLightDirection.Value);  }  if (args.lightEstimation.mainLightIntensityLumens.HasValue) {  myLight.intensity = args.lightEstimation.mainLightIntensityLumens.Value;  }  if (args.lightEstimation.ambientSphericalHarmonics.HasValue) {  RenderSettings.ambientProbe = args.lightEstimation.ambientSphericalHarmonics.Value;  }  }  } |

To get this working you’ll also need to enable light estimation on the ARCameraManager, which you will find as a child in your AR Session Origin object:  
  


That’s it. You’re done. Enjoy your first AR game!

**Step Nine – Experiment**

Okay, now you’re familiar with the building blocks of an AR project, it’s time to experiment. No new AR concepts are required for this, just concepts we’ve discussed used in different ways. Try the following:

* Make the robots react in some way when the camera pans over them. Perhaps they play one of their animations, or have an applied force, or rotate away.
* Using the above as a template, create a new project. When the game starts up, have a single robot prefab already instantiated. Move it to the nearest plane as you pan and move the camera, and place it with a tap.
* Instead of pushing robots around, clone another prefab and have it hurled with force at the asset on the screen.
* Add some movement to the robots through a script so they explore the AR environment you build around them.

These are only suggestions – the key goal of the rest of the time you have available is to become familiar and confident with what each part of the system is doing. AR is only a new way of providing an interface on experiences with which you are already familiar, so understand the new is the important part of being able to replicate and improve upon the old.