How to manage (really) big assets with Addressables

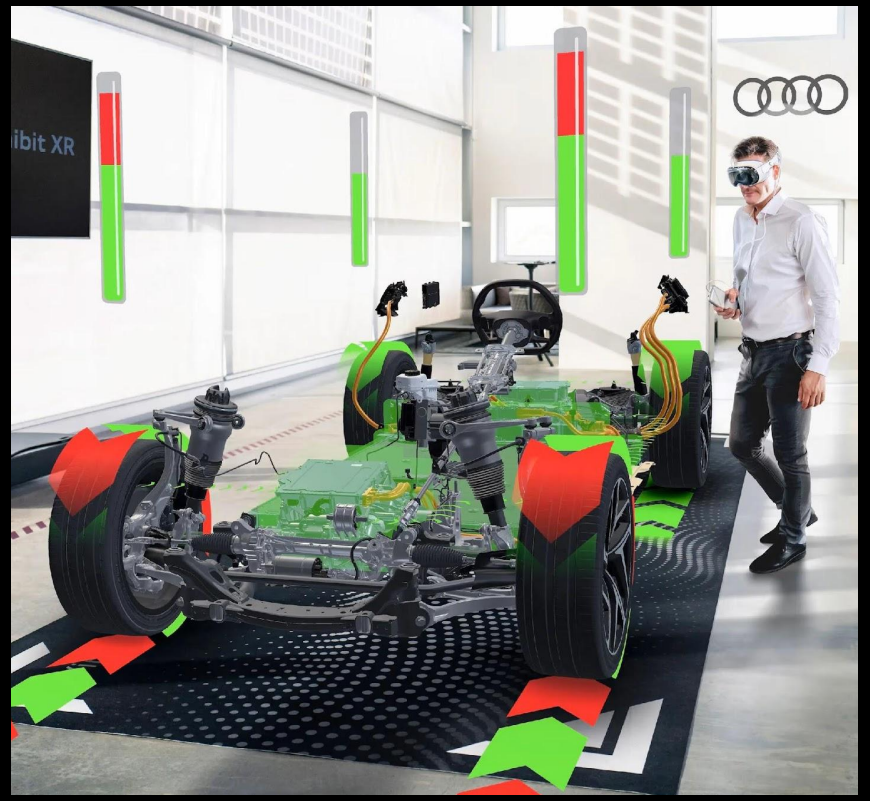
Session 1: An introduction for the industry

The issues we want to overcome

- Industrial assets are often big (multiple Gb)
- *It can cause an issue because we need to load the models in the memory (RAM)*

The issues we want to overcome

- You often need to showcase different models (ex: vehicles)
- The models can be updated regularly by the CAD designers
- *It can cause an issue because we need to load multiples (big) models, and they are often updated*



The issues we want to overcome

- You often present the models on Tablets or autonomous VR Headsets
- *It can cause an issue because these devices have low memory (RAM)*

The issues we want to overcome

- Maybe it's a PiXYZ module that is converting the CAD to FBX or GLB and it's stored in a server

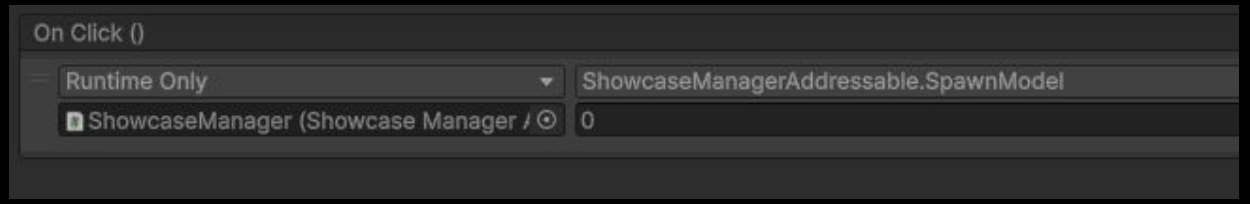
- *It can cause an issue because we need to retrieve those models from the server everytime there is an update*

Our example

- We want a hangar to showcase our vehicles
- We need a simple interface to choose the vehicles we want
- We want to be able to showcase multiples vehicles

The classic way

- 1 prefab = 1 vehicle
- A script that references all the prefabs
- A button per model to spawn them



unity

The code for Prefabs

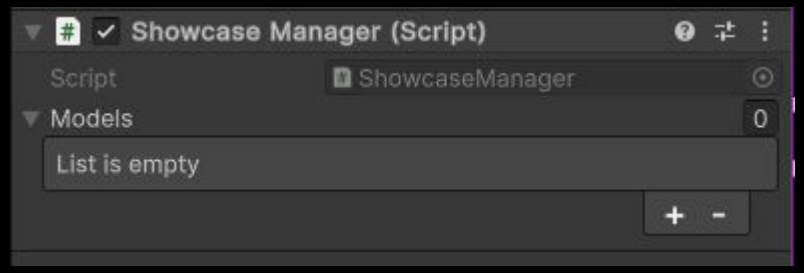
```
public class ShowcaseManager : MonoBehaviour
{
    [SerializeField]
    private List<GameObject> models = new List<GameObject>(); // the list of all references to the prefabs

    private GameObject currentModel; //the current loaded model in the scene


    public void SpawnModel(int id)
    {
        if(currentModel != null)
        {
            Destroy(currentModel); // Destroy old model if it exists
        }
        currentModel = Instantiate(models[id], models[id].transform.position, models[id].transform.rotation); //instantiate the model in the scene
    }

}
```

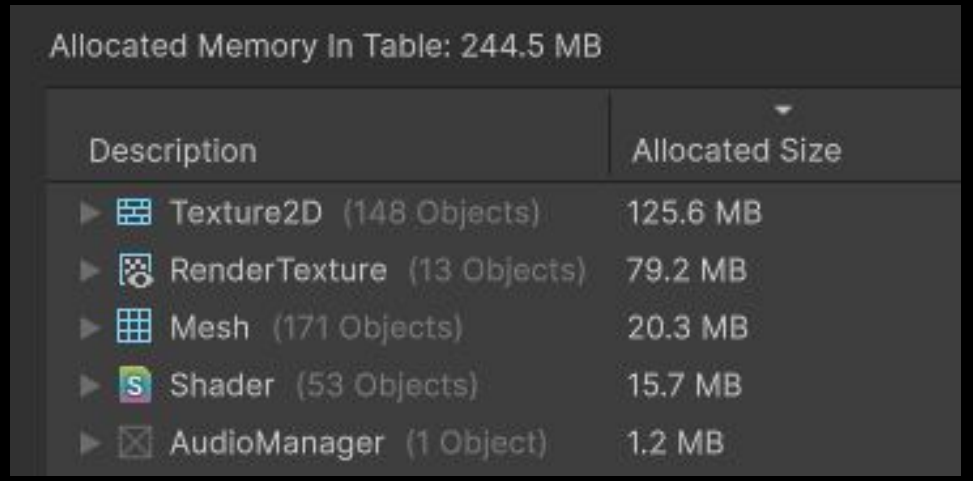
Why it’s a problem ?

EVERY PREFABS ARE LOADED IN THE MEMORY (RAM)

When scene is empty

- Only the Hangar
- Build size : 265MB
- RAM
  - Total : 244MB
  - 80MB = camera view
  - ~150MB = hangar

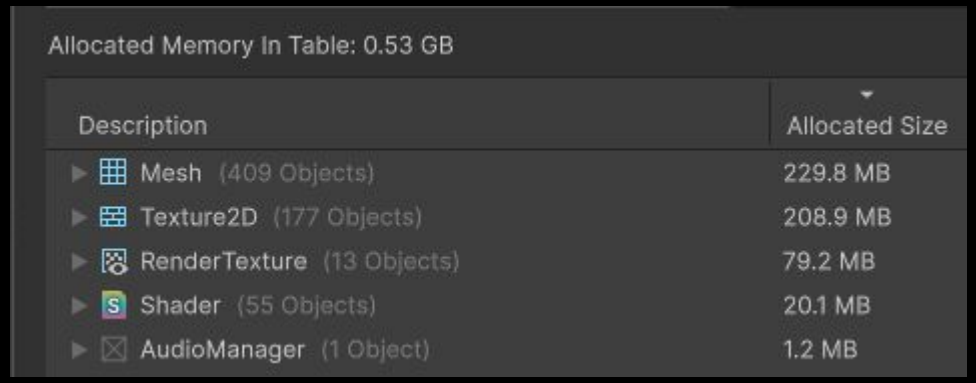




With the prefabs

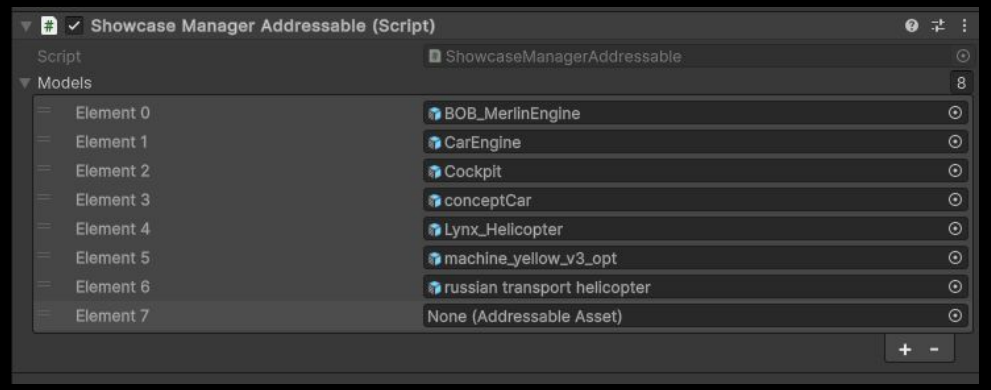
- Added the prefabs in the list
- Build size : 558MB
- RAM
  - Total : 530MB
  - 80MB = camera view
  - ~150MB = hangar
  - ~300Mb = vehicle models

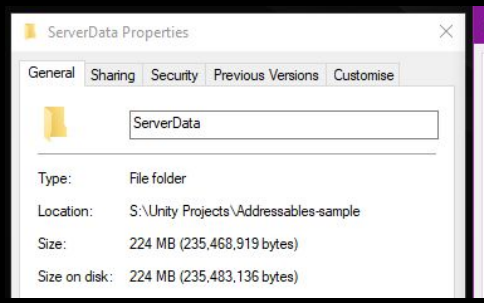
+117%

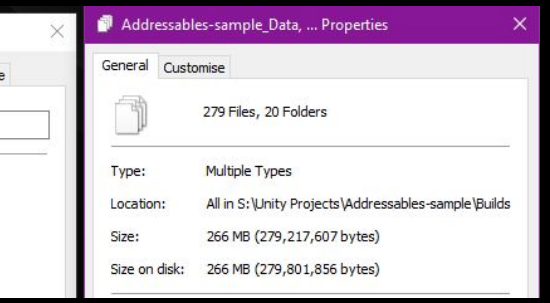


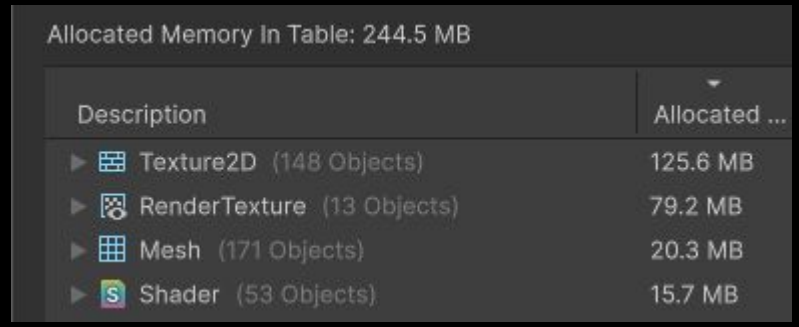
The memory usage with Addressables

- Use addressables instead of prefabs
- Build size
  - Build: 266MB
  - Addressables (bundles): 224MB
- RAM
  - Total: 244 MB
  - 80MB = camera view
  - ~150MB = hangar
  - 0Mb = vehicle models









Let’s do that within the Unity Editor!

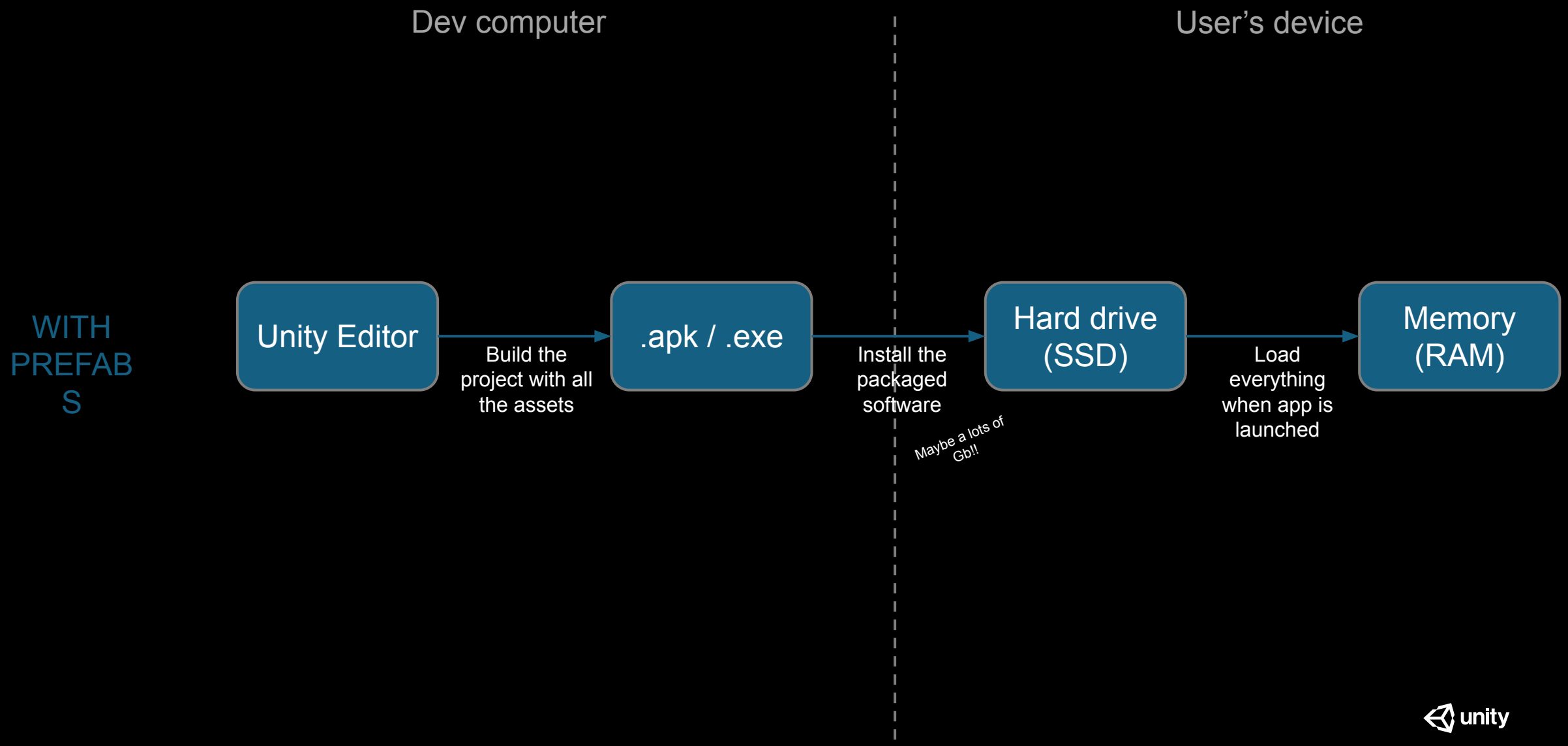
unity

Well Done 

Generative Art – Made with Unity

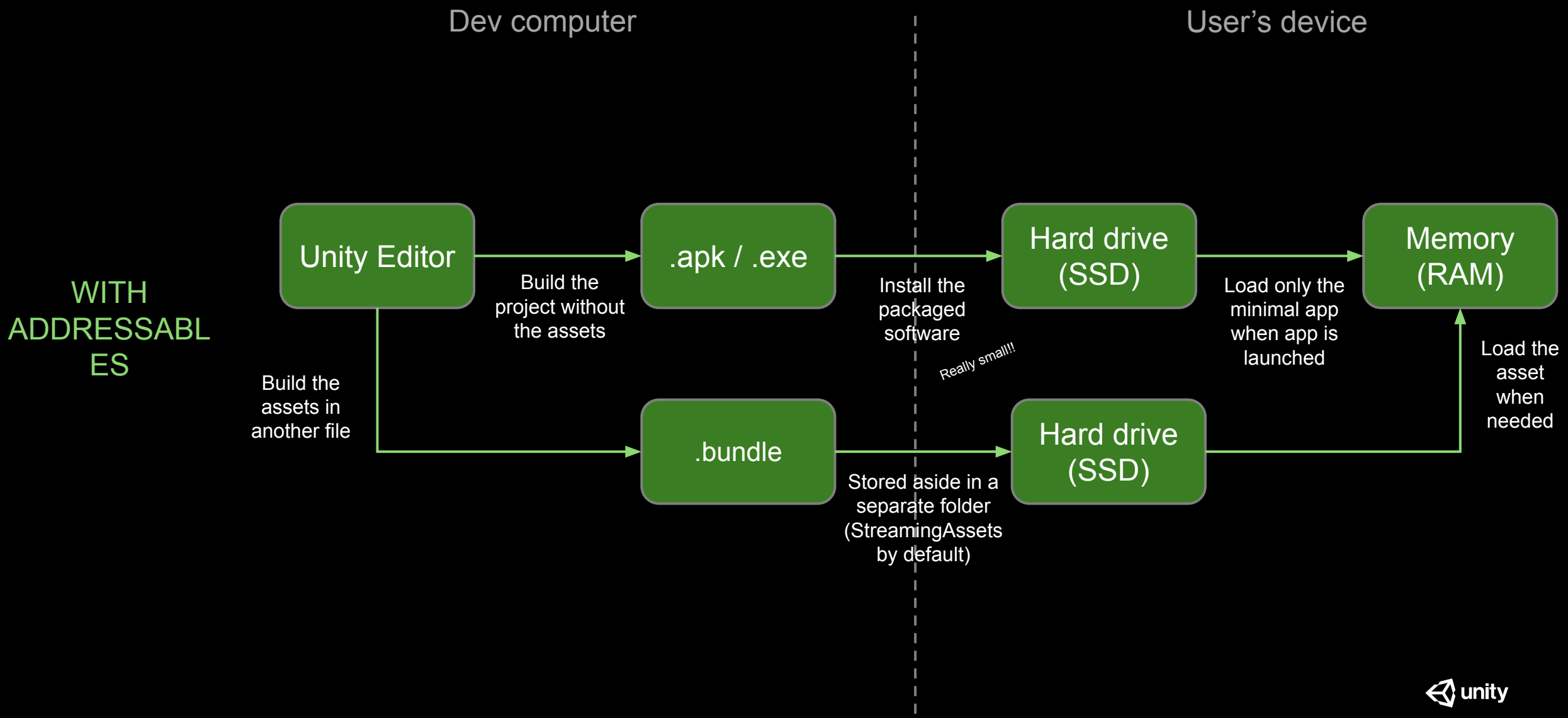
unity

The classic process of building an app



unity

The process of building an app with the adressables

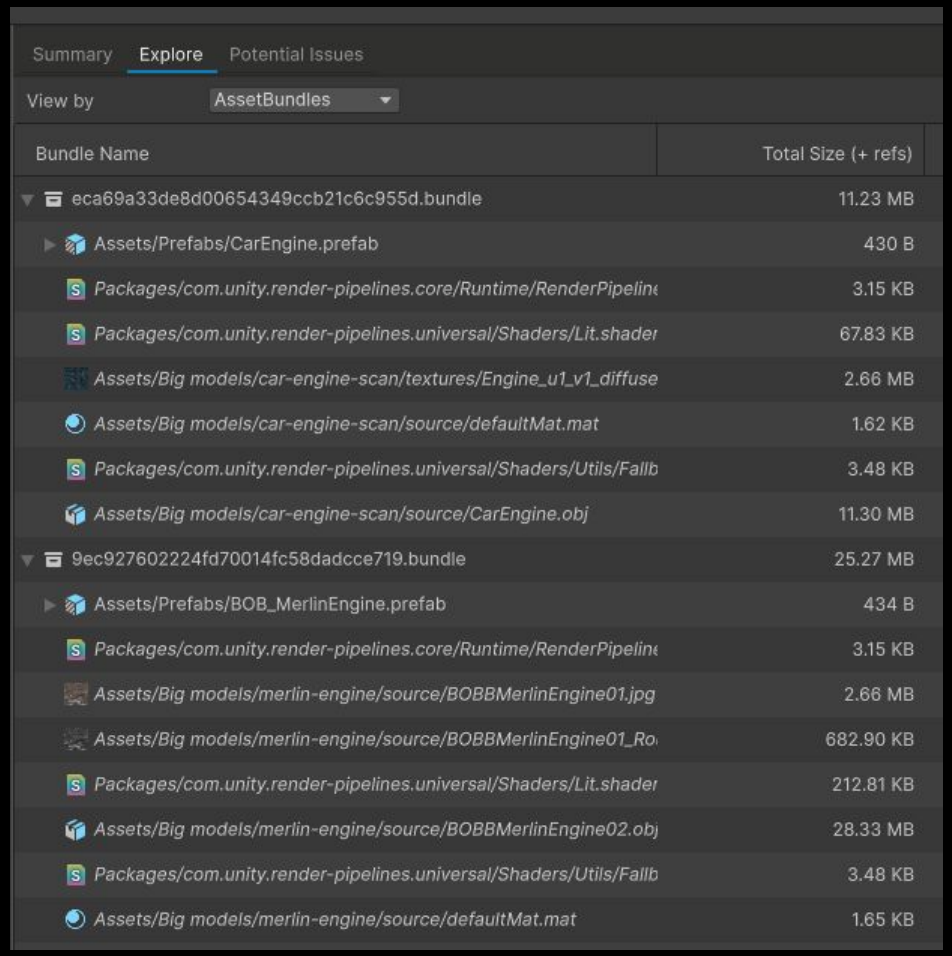


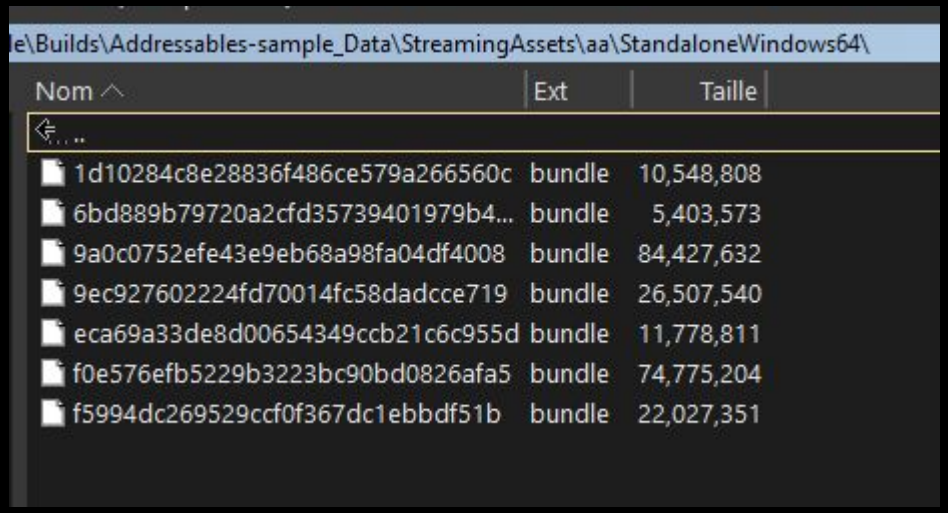
The code for Addressables

```
public class ShowcaseManagerAddressable : MonoBehaviour
{
    [SerializeField]
    private List<AssetReference> models = new List<AssetReference>();

    private AssetReference currentAsset; //the asset of the current loaded model in the scene
    private GameObject currentModel; //the current loaded model in the scene

    0 references
    public void SpawnModel(int id)
    {
        if (currentAsset != null && currentAsset.Asset != null) // Destroy old model if it exists
        {
            Destroy(currentModel); //destroy the game Object in the scene
            currentAsset.ReleaseAsset(); //release the addressable from the memory
        }
        LoadModel(models[id]);
    }
    /// <summary>
    /// Loads the model from the addressable asset reference and instantiates it in the scene.
    /// </summary>
    /// <param name="_reference"></param>
    1 reference
    private async void LoadModel(AssetReference _reference)
    {
        AsyncOperationHandle<GameObject> operation = _reference.LoadAssetAsync<GameObject>(); //load the model from the addressable asset reference
        await operation.Task; //wait for the model to load in the memory

        if (operation.Status != AsyncOperationStatus.Succeeded) //catch the error if the model fails to load
        {
            Debug.LogError($"Failed to load model: {_reference.SubObjectName}");
            return;
        }

        currentAsset = _reference; //set the current model to the loaded model so we can have a ref to it to unload it later

        GameObject loadedGameObject = currentAsset.Asset as GameObject; //convert the loaded model to a game object (because it is a prefab)
        currentModel = Instantiate(loadedGameObject, loadedGameObject.transform.position, loadedGameObject.transform.rotation); //instantiate the model in the sc
    }
```

unity

The Addressables are built as .bundle





unity

Pros / Cons

- Pros:
  - Not every assets are loaded in the memory when the scene loads
  - You can mix prefabs & addressables
  - You are manipulating strings and not assets (CPU & RAM are happy)
  - You choose what is loaded into memory!

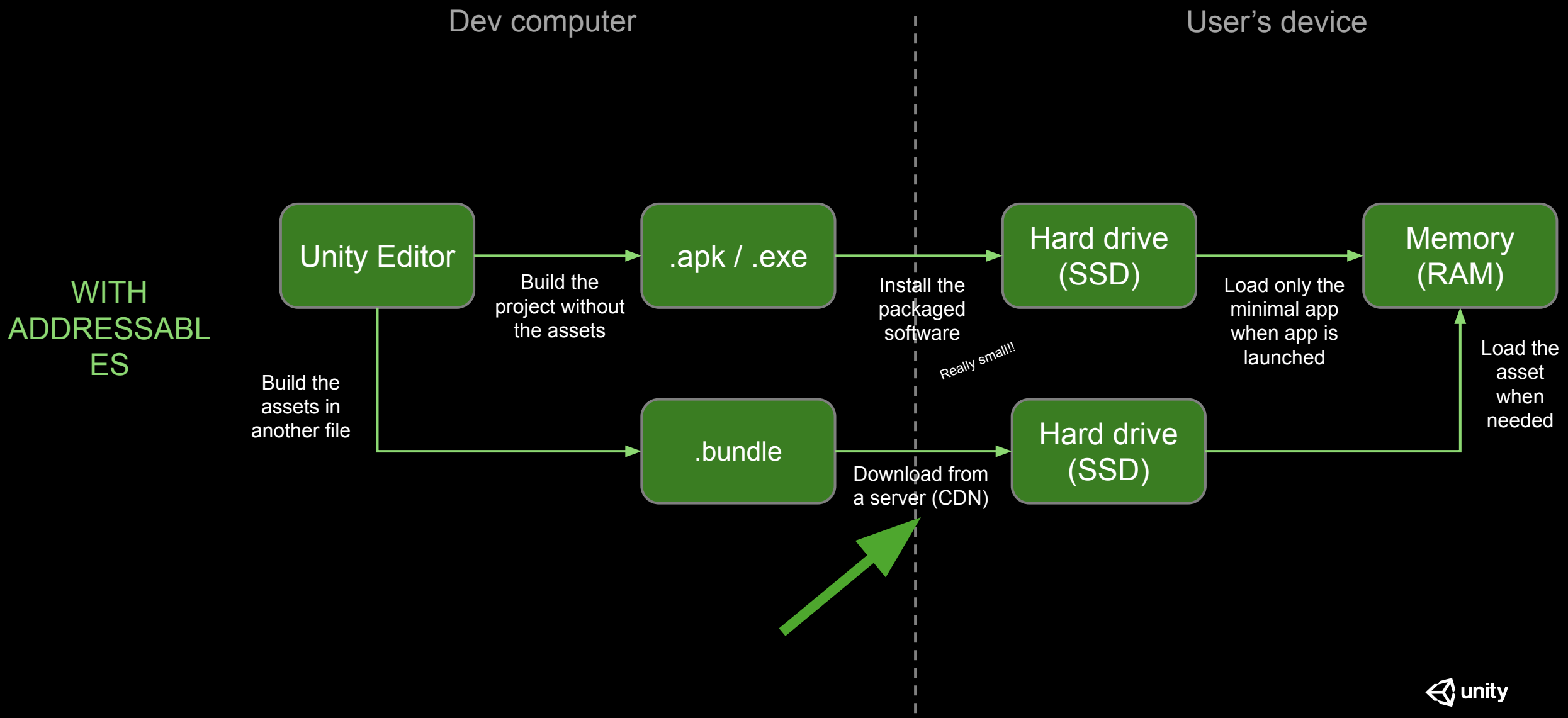
- Cons :
  - You *choose* what is loaded into memory!
  - More code to write.
  - You mustn’t forget to unload
  - You have a new interfaces to learn

WITH GREAT POWER COMES GREAT RESPONSIBILITY

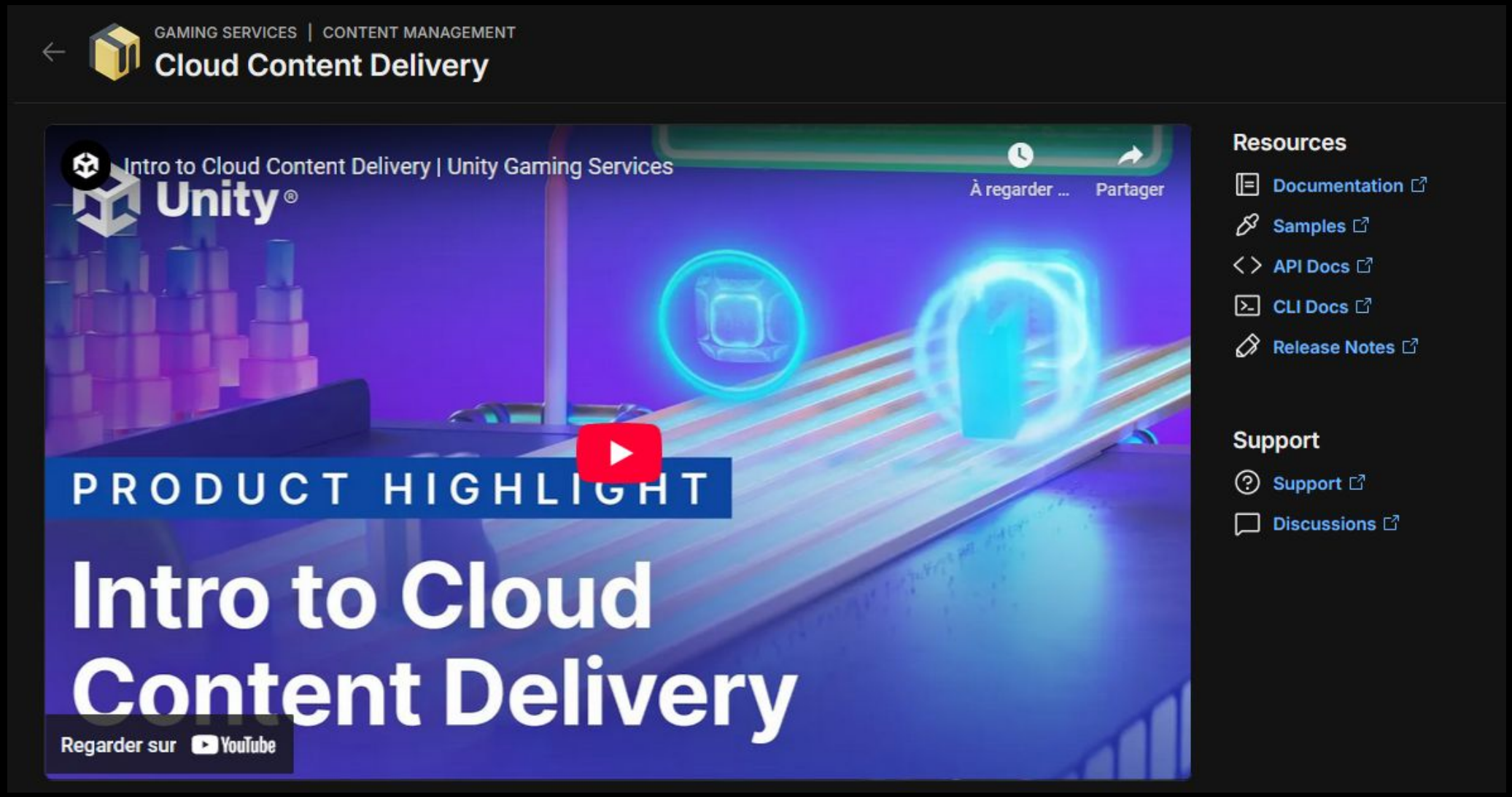
ONE MORE THING

What if my models are constantly updated ?

The addressables can be stored online



Where to store the .bundle ?



unity

Pricing

Cloud Content Delivery			
Cloud Storage		Free	Free
Content Management		Free	Free
Bandwidth (CCD)	ⓘ	50 GB / month	After 50 GB: $0.08 per GB, After 50 TB: $0.06 per GB, After 500 TB: $0.03 per GB

unity

Pros / Cons

- Pros:
  - Build is smaller
  - Assets be replaced online and the app will be updated without creating a new version of the build

- Cons :
  - Build process is a bit more complicated
  - You mustn’t forget to push the .bundle to the server

Let’s do that within the Unity Editor!

unity

Conclusion

- Provides a powerful solution for managing large assets in both industrial and video game projects
- By connecting Addressables to Unity's Cloud Content Delivery you create an efficient update pipeline that improves user experience and simplifies deployment.

Conclusion

- Optimized memory usage by loading only necessary assets when needed
- Remote model updates without rebuilding the entire application
- Smaller initial build sizes
- Flexible asset lifecycle management

- Properly manage asset loading/unloading to prevent memory leaks
- Implement loading systems for download times
- Maintain organized Addressables groups
- Test thoroughly with various loading scenarios

⚠️Warning⚠️

- In our case we force Unity to load from the cloud in order to showcase the feature
- In real-life you want to include the assets in the local build in case the user doesn’t have internet access