ChatGPT

Lens Studio Crash Course

1. Introduction to Lens Studio

What is Lens Studio? Lens Studio is Snap's powerful **augmented reality development platform** for creating Snapchat Lenses. It provides a robust set of features – if you can dream an AR experience, you can build it with Lens Studio [1]. The application runs on Windows 10 (64-bit) or macOS 12+ systems and requires at least a 2.5GHz Core i3/AMD FX 4300 (or Apple M1) with 8GB RAM and a mid-range GPU (e.g. Intel HD 5000 or better) [2]. You can download the latest Lens Studio 5.x from the official Snap AR site and install it like any desktop app. Once installed, launch Lens Studio and ensure you're on the newest version for latest features.

Interface and Workflow: When you first open Lens Studio, you'll be greeted by the Home Screen (Home Page) which offers templates, tutorials, and the new **Lens Studio AI** wizard (which can generate starter Lenses via text prompts) [3] [4]. The interface is composed of multiple **modular panels** [5] that should feel familiar if you've used game engines like Unity. Key UI elements include:

- **Scene Panel:** a 3D viewport to place and manipulate objects (move, rotate, scale) in your Lens scene [6]. This panel has gizmo tools for translation/rotation/scale and camera navigation controls.
- **Scene Hierarchy Panel:** a tree view of all objects in the scene (like layers). It shows parent-child relationships and order (which can affect render order for 2D elements) [7] [8].
- **Inspector Panel:** shows properties of the selected object or asset. Here you adjust components (e.g. transform values, material settings, script parameters, etc.).
- **Asset Browser (Resources) Panel:** where you import and manage assets (images, models, sounds, etc.). You can add new resources or use the built-in Asset Library to grab pre-made assets.
- **Preview Panel:** a live preview of your Lens output as it would appear in Snapchat's camera [9]. By default, it uses a sample video feed or 3D model (e.g. a face model) to simulate a user. Changes in your scene update here in real-time [10]. You can switch the preview between front-camera (selfie) and rear-camera views or test on different device aspect ratios. You can even open **multiple Preview panels** to test various scenarios simultaneously (e.g. different faces or environments) [11].
- **Logger Panel:** a console that shows debug messages, errors, and warnings. It's useful for debugging scripts and tracking performance logs [12].
- **Toolbar:** at the top, with useful actions (e.g. undo/redo, play/stop, etc.) and a **Preview Lens** button to test on a real device.
- **Workspaces:** Lens Studio provides preset workspaces (e.g. Default, Animation, Scripting) which rearrange panels for those workflows. You can customize and save your own layout. If you ever get lost, use **Window > Default Layout** to reset the interface [13].

Take a moment to familiarize yourself with the interface. You can drag panels to dock or float them, and **multiple panels of the same type** can be opened (e.g. two Scene panels or Inspectors) to edit different things in parallel [14]. The UI is highly configurable to your needs.

System Setup and Project Creation: To start a project, you can select a template on the Home Screen or create a **New Project** (empty scene). Lens Studio will create a default scene with a **Camera** and some helper objects depending on the template. It's recommended to log in with your Snapchat account in Lens Studio for easier device preview pairing and publishing later. As you build, save your project (.lsproj) frequently.

Tip: Lens Studio's **community** is very active – consider joining the official Lens Studio Discord for help and inspiration [15]. Also, the **Asset Library** (accessible via the panel or + menu) contains tons of free assets, templates, and even machine learning models to speed up your creation.

2. Lens Types (Face, World, Full-Screen, Segmentation)

Snapchat Lenses generally fall into a few broad **types of AR experiences**, which Lens Studio supports:

- **Face Lenses (Face Filters):** These are lenses that use the **front camera** and attach effects to a user's face. Lens Studio's face tracking can find one or multiple faces and apply effects like masks, hats, makeup, distortions, etc. A face lens follows the user's head movement and facial expressions. *Definition:* "When using a phone's front camera, there are AR effects attached to faces found" [16]. Lens Studio provides many face-specific templates (e.g. Face Paint, Face Mask, Face Inset) and built-in face effects (retouching, face stretch, etc.). With face lenses, you can achieve the classic Snapchat selfie filters – think dog ears, face swap, character masks, beautification filters, etc.

- **World Lenses (World Effects):** These use the **rear camera** and place content in the environment around the user. World Lenses are anchored to the world (often the ground or a surface) and viewed in **6 degrees-of-freedom** so they stay in a fixed real-world position as you move the camera [16]. Examples include placing a 3D character on your desk or augmenting a landmark. Lens Studio supports world tracking via SLAM (Simultaneous Localization and Mapping) – it can track the device motion and **anchor objects on planes or in mid-air**. It also offers **surface detection** and an optional **World Mesh** feature that reconstructs the geometry of the environment for more realistic placement and occlusion. In fact, Lens Studio's **Device Tracking** component in *World* mode enables advanced world AR features like instant surface hit-testing, **depth sensing**, and world meshing [17]. Use world lenses for AR portals, games on a table, or placing life-size objects in a scene.

- **Full-Frame (Full-Screen) Lenses:** We use this term for lenses that **affect the entire camera feed output**, rather than attaching to a specific face or world object. These often involve **2D overlays, color filters, and post-processing effects** that cover the full screen. For example, a lens that tints the screen a certain color, adds a vignette, or distorts the entire image would be a full-frame effect. Technically, these are achieved with **Screen Space** objects or **Post Effect** components in Lens Studio. A common approach is using an **Orthographic camera** with full-screen Image or Post Effect to layer filters over the view. Many "style" lenses (like VHS camera look, color gradients, or weather effects that cover the whole view) fall in this category. Full-screen lenses often work on both cameras since they're not tied to world or face anchors.

- **Segmentation Lenses:** Segmentation is an advanced feature that allows "cutting out" parts of the camera image (like a green screen effect). Snapchat provides **ML-based segmentation** to isolate elements such as the user's **portrait (body)**, **hair**, **sky**, etc. A segmentation lens uses these masks to apply effects only to certain regions – e.g. put a different background behind people, or recolor

someone's hair. Lens Studio includes several **Segmentation Texture** assets (Portrait Background, Portrait Hair, Portrait Head, Sky, etc.) which act as real-time masks [18] [19]. For example, the **Portrait Segmentation** texture masks out the user's background and lets you replace it with an image or video [20] [21] (like a virtual backdrop – think of Zoom background replacement). You can also invert the mask to affect only the user. Segmentation lenses are extremely popular for backdrop replacement, AR try-on (isolating clothing), or creative effects (putting a user into a scene). In Lens Studio, you simply add a Segmentation Texture asset, choose the type (e.g. Portrait vs. Sky), then assign it as a **Mask Texture** on a Camera or material [22]. The result is that only certain parts of the scene render, achieving the cut-out effect. You can further refine masks with options like feathering and edge refinement for better quality [23].

Keep in mind these lens types are not mutually exclusive – **one Lens can combine multiple tracking types**. Lens Studio even has a guide for **Combining World and Face** effects in one project [24] [25]. For instance, a lens could put a virtual hat on your head (face lens) while also spawning a 3D character next to you (world lens). Technically, a single project can include face trackers, world anchors, and full-screen filters all together. Just be cautious with complexity and performance if you do too much at once.

Finally, note that Snapchat also recognizes **Marker Lenses** as a category: these are a special case of world lens that triggers when the camera sees a specific image or Snapcode. Lens Studio's **Marker Tracking** allows you to track content to a **unique physical image** – for example, overlay an AR animation on a poster or a QR code [26]. When the user points the camera at the target image, the AR content appears locked onto it. You can create Marker Lenses by providing a reference image for tracking; Snap even allows using Snapcodes as markers [27]. If your project calls for an AR experience tied to printed media or logos, marker tracking is the feature to use.

Summary: Lens Studio supports all major AR lens types – selfie filters, world-anchored AR, full-screen effects, background segmentation, and more. Understanding the lens type helps determine which tracking components and templates to start with. For a beginner, it's easiest to choose **Face Lens** or **World Lens** as a foundation and then possibly layer a full-screen effect or segmentation on top. The Home Screen's "I want to build for..." section can guide you to relevant starter templates for face, world, etc., along with sample projects and tutorials [28] [29].

3. Object Types in Lens Studio

Lens Studio uses a scene graph of **objects** (nodes) that have components to define their behavior/appearance. When you click the **"+" (Add New Object)** button in the Scene panel, you'll see many object types you can add [30]. Key categories include:

- **Empty Objects (Scene Object):** A null container or empty locator. It has a Transform and can be used as a parent to group other objects or to serve as an anchor. This is equivalent to an empty GameObject in Unity – useful for organization or as a pivot.

- **3D Objects (Mesh, 3D Models):** These are objects that have a **Mesh component or Mesh Visual**, meaning they render a 3D model. You can import 3D models (FBX, OBJ, GLTF) into the Asset Browser and then create a Mesh object in the scene with that model. 3D objects can also include primitives (like sphere, plane) and surface materials. When you add a **Mesh Visual** object from the menu, Lens Studio inserts a default shape you can replace with your model. 3D objects exist in world coordinates

(affected by perspective camera). You can attach various components to them: e.g. **Collider or RigidBody** (for physics), **VFX** (particle effects), or **scripts** for interaction. They can also be rigged (with joints for body tracking) or skinned to follow face shapes, etc. Lens Studio's **3D** asset pipeline supports PBR materials, animations (you can import animation clips or use the Animation Player component), and even rigged characters like Bitmoji avatars.

- **2D Objects (Screen Image, Text, etc.):** These are flat objects that render to the screen space (HUD or overlay). For example, **Screen Image** adds a 2D image UI element that you can anchor to the screen or camera view. **Screen Text** lets you display text labels. These objects use a **Screen Transform** component instead of a regular 3D transform, meaning you position them in 2D coordinates (pixels or relative units) on the screen or UI canvas. Lens Studio's UI system includes **Screen Regions** (to adapt to different phone aspect ratios) and a **Canvas** component (to mix 2D UI within a 3D scene) [31]. Use 2D objects for UI elements like buttons, score counters, or full-screen overlays (e.g. a frame or border graphic). Note that 2D objects by default render on top unless you adjust hierarchy order or use the Orthographic camera's layers.

- **Camera:** Cameras determine what is rendered and provide render targets. A Lens requires at least one **Camera** to render the scene to the output. By default, new projects include a camera. You can add additional cameras via `+ -> Camera` [32]. Lens Studio supports **Perspective** and **Orthographic** cameras [33]. Perspective cameras mimic real-world perspective (for 3D content), while Orthographic cameras have no depth foreshortening (useful for 2D overlays or rendering text/icons that shouldn't scale with distance) [34]. Cameras have properties like layers (to control what renders), depth order, **Mask Texture** (for segmentation mask input) [35], and so on. You might use multiple cameras if you want to render certain objects separately (for example, one camera rendering background with segmentation, another for foreground elements). However, for most lenses the single default camera suffices after configuring its tracking (face or world mode). Remember to set **"Lens Works On"** in Project Settings to front, rear, or both cameras appropriately for your lens use-case.

- **Lights:** Lighting is critical for 3D object realism. Lens Studio supports several light types. By default, a new project might include an ambient light or a single directional light. You can add a **Light** via `+ -> Light` (underneath the objects menu). In the Inspector you can choose the light type: **Directional** (sunlight, global direction, affects all objects), **Point** (omnidirectional light from a point source, with range), or **Spot** (if supported – LS documentation mainly references directional and point lights). Lights can cast **shadows** (Directional lights have a "Shadows" checkbox – only one directional shadow is allowed per lens) [36]. The lighting system is simplified for performance, but you can achieve good results with environment light (via **Environment Maps** for image-based lighting) and the physically-based materials. For example, to get shadows on the ground, enable shadows on a Directional Light and ensure your objects/materials support receiving shadows (use a PBR material and have a mesh receiving shadow). Proper lighting will make your AR objects feel anchored in the scene (e.g. correct shading and shadows on the ground).

- **VFX and Particles:** Lens Studio has a **VFX Editor (Particle System)** for advanced visual effects like smoke, sparks, and weather. You can add a **Particle System** object or use the **VFX Graph** to create custom particle behaviors. This is more advanced, but note that in the Objects menu you might see things like **"Post Effect"**, **"Particle System"**, or **"Emitter."** Post Effects are full-screen shader effects (color correction, blur, LUTs) that you apply to the camera image [30]. They come as objects with an

attached PostEffect component and material. Particle Systems emit many sprite particles and can be configured in the VFX graph (with forces, behaviors, etc.). The **Graphics, Materials and Particles** section of the docs has a wealth of info on customizing these for creative effects.

- **Animated Objects:** These can be regular 2D/3D objects with an **Animation component** or **Skeletal animation**. For example, if you import a rigged FBX with animation, Lens Studio will create an **Animation Asset** and you use an **AnimationPlayer** component on a scene object to play it [37]. The AnimationPlayer can target a specific object (e.g. a imported 3D character) and play its clip. There are also **Helper objects** like **Head Attached Object** (a special object that auto-binds to a face head position) [38] or **Body Attachment Point** etc., which automatically follow tracked joints.

- **Custom Components/Prefabs:** Lens Studio allows packaging objects into **Prefabs** (reusable object bundles). A prefab is essentially an object (or group) saved as an asset that can be instantiated multiple times. If you find yourself duplicating an object setup, consider making it a Prefab – then editing the prefab asset updates all instances [39] [40]. Prefabs are also a convenient way to **organize complex Lenses into sections** (you can load/unload prefab content or spawn prefabs via script at runtime). Many assets from the Asset Library import as prefabs which you drag into the scene. You can always **unpack a prefab** to tweak its internals [41]. Prefabs are great for managing repeating elements (like multiple enemies in a game lens) or optional scenes (like a separate UI overlay).

- **UI Widgets:** Beyond basic Screen Images and Text, Lens Studio offers higher-level UI **Widgets** as **Custom Components** (available via the Asset Library). For example, there are built-in UI components for things like **Scroll Panels**, **Sliders, Buttons**, **Image Carousel**, and even a **Camera Roll picker** [42] [43]. These come as packages you import and instantiate. They often include scripts to handle interaction. If your lens needs a sophisticated UI (like a scrolling menu or a multi-select carousel), leverage these widgets instead of building from scratch. They follow responsive design to adapt across devices.

To add any object, click the **"+"** in the Scene panel or right-click in the Scene Hierarchy. Objects with preset components are listed (Camera, Light, Face Effects, 3D Objects, 2D Objects, etc.) [30]. Choose what you need and it will appear in the hierarchy. You can then customize it in the Inspector.

Hierarchy Tips: You can drag objects in the Scene Hierarchy to parent/unparent them (e.g. make an object a child of the head tracker so it follows the head) [44]. The order of objects can matter for 2D rendering (lower in list = rendered on top for screen objects) [45]. Use grouping empties to keep things tidy. For instance, group all UI under a "UI" object, or all environmental props under a "WorldContent" object.

Lens Studio's object model is quite flexible. It's worth exploring the **Objects** panel to see the full list: you'll find things like **Face Effects** (which are essentially pre-configured objects with components for specific face behaviors: e.g. Face Mesh, Face Retouch, Eye Color, etc.), **Surface (Plane) Tracker**, **Hand Joints**, **Particle Emitter**, **Billboard** (always faces camera), **Speaker** (audio), and more. Each item in the add menu either creates a single object with certain components or a small hierarchy of objects ready to go.

4. Scripting in Lens Studio (JavaScript & TypeScript)

While you can create simple Lenses with just components and visual tools, **scripting** unlocks the full power of Lens Studio. Scripting lets you add custom interactions, dynamic behaviors, game logic, and integrate external data. Lens Studio supports scripts in **JavaScript (ES6)** or **TypeScript** – you can even mix both in a project. Under the hood, scripts run in a sandboxed JavaScript engine within the lens.

Creating and Attaching Scripts: To add a script, create a **Script asset** in the Asset Browser (`+ -> Script` gives a .js file) [46]. Write your code in Lens Studio's built-in editor (double-click the script asset to open the **Script Editor** [47]) or in your own IDE (Lens Studio can sync with VS Code via an extension [48]). Once you have a script asset, you attach it to the scene by adding a **Script Component** on an object and assigning that script asset to it [49]. The Script Component in the Inspector lets you bind your script to certain events (more on events soon). You can add multiple Script Components to different objects or the same object as needed.

Lens Studio Script Basics: Scripts are written in plain JavaScript/TypeScript, but there are some specifics to the Lens Studio environment [50]. Each script file is essentially a module that Lens Studio will execute. You have access to a rich **Lens Studio API** (for example, to find objects, control components, play animations, etc.) and a limited global environment (for security, some JS features or libraries might not be available).

One important concept: **Event-driven architecture.** In Lens Studio, scripts **don't have a continuous main loop** that you write yourself; instead, they respond to events (like "Lens initialized", "per frame update", "user tapped the screen", "face opened mouth", etc.). By default, when you add a script component, it auto-binds to the **"On Awake"** event (which triggers when the lens starts or when the object is enabled) [51] [52]. You can change or add events in the Inspector for the Script Component or via code. For example, in code you might do:

```
var updateEvent = script.createEvent('UpdateEvent');
updateEvent.bind(function() {
    // code that runs every frame
});
```

This uses the Lens Studio API `script.createEvent` to register an **Update event** callback (executed every frame at ~30fps) [53]. Similarly, you could create a **TapEvent** to respond to user taps, etc. Most commonly, you'll set up events using the Inspector UI: select the Script Component and choose an **Event** from a dropdown (like "Tapped" on some object, "Mouth Opened", "Face Found", etc.) and point it to a function in your script (annotated with `//@input` or `//@onEvent` as needed). The **Script Events** guide enumerates many available events, categorized into **Scene Events** (OnAwake, OnStart, OnUpdate, OnDestroy – typical lifecycle), **Touch Events** (TouchStart/End, Tap, etc.), **Face Events** (Blink, MouthOpened, etc.), **Camera Events**, and others [54]. Essentially, you write **event handler functions** and bind them to Lens triggers.

Example: Suppose you want an object to bob up and down continuously. You could use an UpdateEvent to adjust its Y position each frame (sine wave motion). Or if you want to play a sound when the user taps the screen, you'd use a TapEvent. If you want something to happen 3 seconds after lens starts, you might use a

delayed callback or a separate **Time Event**. The event system is very flexible. The key is to **think in terms of responding to triggers** rather than an arbitrary game loop.

Common Scripting Patterns: A few patterns are especially useful in Lens Studio scripting:

- **Script Setup (On Awake/On Start):** Initialize your variables, find scene objects, and configure things in an On Awake or On Start handler. On Awake runs once when the script is loaded [55]; On Start runs on the first frame after all Awakes (useful when you need other objects initialized first) [56]. Example: find a reference to an object by name: `var obj = scene.root.find("ObjectName")` (Lens Studio provides a `global.scene` for root).

- **Update Loop:** Use `script.createEvent('UpdateEvent')` to do something every frame [57]. Keep it lightweight (for heavy per-frame work, consider if you can use built-in components or VFX Graph instead). For instance, updating an object's position for an animation or checking a condition each frame can go here.

- **User Interaction via** Behavior Script**:** If you're not comfortable writing your own event code, Lens Studio offers a built-in **Behavior Script** component. This is a custom script (available in the Asset Library) that lets you configure "When X happens, do Y" through the UI – essentially an **event-response system without coding** [58]. For example, "When Screen is Tapped, Play Animation on Object" can be set up with Behavior scripts. It covers many triggers (face events, time delay, triggers when entering/exiting camera view, etc.) and responses (enable object, play VFX, etc.). Behavior scripts are great for simple interactions and rapid prototyping. Under the hood, they are scripts using the same APIs, but you just fill out fields in the Inspector. If you need logic beyond what Behavior offers, you'll move to custom scripting.

- **Tweening and Animations:** Another helper is the **Tween Manager** (also available via Asset Library) [58]. This provides convenient functions to animate properties over time (e.g. smoothly move an object, fade opacity, etc.) without writing the interpolation code manually. You create a tween (say move from point A to B in 2 seconds) and it handles the update events for you. Using Tweens can simplify your script if you want to do timed animations or easings.

- **Custom Script Inputs:** You can expose parameters on your scripts that are adjustable in the Inspector using special comments. For example, writing `//@input float speed = 1.0` at the top of your script will create a "Speed" field in the Script Component UI [59] [60]. This way, you can tune values without modifying code, and even allow non-programmers on your team to tweak behaviors. You can have inputs of types: bool, int, float, vec3, color, Asset references, etc. These show up under **"Custom Script UI"** in Inspector. It's best practice to use script inputs for any tunable parameters (like sensitivity of an interaction, movement speed, etc.) so that they can be adjusted easily and even animated or linked to UI widgets.

Lens Studio uses a **component-based approach** similar to Unity. That means instead of one giant script, you might attach smaller scripts to relevant objects. For instance, attach a "Rotate continually" script to an object that needs to spin, and attach a separate "Handle tap" script to an object that should respond to taps. They will run independently. Keep in mind script execution order follows the Scene Hierarchy order for OnStart and Update if not otherwise specified [61] (generally not an issue unless you have interdependent scripts).

Scripting Language and API: The Lens Studio scripting API is extensive – you can manipulate objects, components, materials, etc. through script. A few examples: `script.getSceneObject()` gets the object this script is attached to; from there `getTransform()` gives its Transform component to change position. There are global objects like `global.scene` to access other parts of the scene, or `global.tweenManager` if using tweens. Check the **API Reference** panel in Lens Studio (or the online docs) for classes like `Component.AudioComponent` or `FaceTracking` etc. Use the API reference to find what you can do – for example, the `AudioComponent` has methods like `.play()` to play a sound [62], the `Tween` class has methods to start/stop tweens, etc.

Lens Studio now also supports **TypeScript** if you prefer static typing. You can toggle a script asset to TS mode and even get code completion with the VS Code Lens Studio extension [63]. This is great for larger projects to catch errors early. Under the hood, LS will transpile TS to JS when building the lens. The API is the same; you just get type safety.

Event Best-Practices: It's often useful to use certain events for specific purposes: - Use **On Awake** to setup references (but not to interoperate between scripts, as others may not be awake yet). - Use **On Start** to do things like start animations or sound (when you need everything initialized). - Use **On Update** sparingly – if you just need something to happen after a delay, consider a **DelayedCallback** (Lens Studio has `script.createEvent("DelayedCallbackEvent")` as well) or use the built-in Tween/animation events instead of a manual update loop. - Use **custom triggers**: You can emit your own events or simply call functions on other scripts via `scriptObject.sendMessage()` or by obtaining a reference to another script via `getComponent("Component.ScriptComponent")`.

One of the most powerful aspects of Lens Studio scripting is integration with other features: e.g., you can call **Lens Cloud APIs** or **Remote APIs** via script to fetch live data (more on that in section 13), or control **SnapML** outputs (if an ML model identifies an object, your script can respond and spawn something). The possibilities are wide open.

To debug scripts, use `print()` statements to log to the Logger panel (there's no step debugger, so logging is your friend). The Logger will show your prints and any runtime errors with line numbers. You can also use the **Inspector** while running to watch values if you expose them as script inputs and tweak in real-time.

In summary, **scripting allows you to implement custom logic** that goes beyond static effects. Lens Studio's JS/TS environment is event-driven and optimized for the 30fps, mobile context. Keep scripts efficient and lean (complex math or heavy loops can drop frame rates). Use built-in components when possible (e.g. don't script physics if you can use the physics engine; don't manually interpolate if you can use Tween). And remember to leverage the helper scripts Snap provides (Behavior scripts for quick triggers, Tween for animations, etc.) to accelerate development [58].

5. Visual Scripting (Script Graph)

If coding isn't your forte or you prefer a more visual approach, Lens Studio (in recent 5.x versions) introduced a **Visual Scripting** feature, currently in Beta [64] [65]. This allows you to create logic using a **node-based graph**, connecting **blocks (nodes)** that represent events, functions, and data flow, instead of writing text code. It's conceptually similar to Blueprint in Unreal Engine or Visual Scripting in Unity.

To use Visual Scripting, you would add a **Script Graph** asset and open it to edit a flow chart of nodes. Each node could represent things like "On Lens Turn On event" or "If a value is true then…", "Set object's position", etc. You then add a **Visual Script Component** to an object and assign that graph (similar to attaching a JS script). The graph will run according to its event nodes and connections.

When to prefer Visual Scripting: This approach is great for designers or beginners who want to add interactivity without writing JavaScript syntax. For example, turning an object on/off on a trigger can be done by connecting an "Event: Tap" node to a "Set Active = false" node for the object. It's also useful for simple arithmetic or comparisons driving lens behavior (without worrying about code typos). Visual Scripting can make the logic more **observable** at a glance since you see boxes and arrows representing program flow.

However, since Visual Scripting in Lens Studio is relatively new (and still labeled **[Beta]** in early 2026), it may have some limitations. At the time of Lens Studio 5.0, the visual scripting feature was not fully functional for production use [66]. Assume improvements are ongoing – check the latest release notes to see its status. It's wise to **keep backup logic in script form** or test thoroughly if you build a critical feature in the Script Graph. As of now, most experienced creators use text scripting for complex logic due to the maturity and performance, but visual scripts can be handy for prototyping or for very straightforward interactions.

As Visual Scripting matures, you might prefer it for **state machines**, simple UI logic, or effects sequencing, especially if you're collaborating with non-coders. Snap's docs also provide a reference of **Visual Scripting Nodes** available [67] [68] so you know what building blocks you have. These include nodes for math operations, time delays, event listeners, etc.

Bottom line: If you find coding daunting, give the Visual Scripting graph a try – you can always mix and match (a lens can have some objects using script graphs and others using JS scripts). Use whichever approach you're most comfortable and productive with. The goal is to achieve the interactive behavior you want, either by visually linking nodes or writing code. And because it's in beta, if you hit a roadblock with visual nodes, don't hesitate to implement that part in a traditional script instead.

(Note: Visual Scripting is evolving; keep an eye on the Lens Studio release notes and documentation for updates on new nodes and stability improvements.)

6. Tracking Technologies (Face, Body, Hand, Marker, World Tracking)

Snapchat Lenses are powerful because of the **tracking technologies** that allow digital effects to realistically follow real-world features. Lens Studio exposes all of Snapchat's tracking capabilities:

- **Face Tracking:** The ability to detect and track faces (up to 6 faces at once on modern devices). This is the foundation of face lenses. Lens Studio provides a high-level **Face Controller** (the "Head Binder" object) that you can use to attach objects to a face. But it goes much deeper – the SDK can track **facial landmarks** (eyes, nose, mouth corners, etc.), **expressions** (smile, eyebrow raises), and even a full 3D **face mesh** (a 3D model that deforms to match the user's face shape). The Face Tracking API enables effects like Face Mask (texturing the face or replacing it), Face Inset (cutting and moving facial features), and triggering events on expressions (e.g. Mouth Open event to do something when user opens their mouth [69]). Snap offers many **Face Effects** components so you rarely need to

manually handle the math – e.g. add a **Face Retouch** component to smooth skin [70], or a **Face Landmark** to get points like pupil positions. These are available under the **Face Effects** category when adding objects. In script, you can also access `FaceTracking` components to get live data (like `FaceTracking.face(0).mouth.openness` etc.). For multi-face lenses, each face is indexed (0,1,2,...). Many face effect components have a "Face Index" property to target a specific face [71]. For example, if you want glasses to only appear on the second face detected, you'd set Face Index = 1. Lens Studio's face tracking is very robust and works across skin tones, various angles, and supports **facial occluders** for when the face is partially hidden (so effects don't jitter).

- **Body Tracking:** Snap has introduced full-body tracking capabilities. This allows you to track a person's **skeletal joints in 3D** (like head, shoulders, elbows, wrists, hips, knees, etc.). In Lens Studio, you can add a **3D Body Tracking** object which provides a rig (skeleton) that matches the user's body pose [72]. You can then attach 3D objects to these joints or retarget a 3D character to follow the user's movements. Body tracking supports multiple people as well – it can track **one or more bodies** simultaneously [73]. For simpler use cases or older devices, there's also an **Upper Body** mode (tracking just the torso and arms). Full body tracking is great for lenses that dress the user in virtual outfits, insert a 3D avatar that mimics the user, or games where your body motion controls something. Implementing it involves adding the Body Tracking component, which creates a "stick figure" rig. You then either use the provided attachment points (the component exposes slots for each joint where you can assign an object) or you parent objects under the specific joint bones in the hierarchy. For example, to put a virtual cape on the user, attach it to the spine or shoulders joint of the body tracker. Snap even provides a **reference FBX of the skeleton** (in A-pose or T-pose) so you can rig your custom models to the correct joint names [74]. Body tracking is **3D**, meaning if the user turns around, the joints rotate accordingly (supported via the back camera on modern devices). The tracking is fairly good for broad movements, though fine details like finger tracking are separate (see Hand Tracking). Performance-wise, full body tracking uses more processing, so test on your target devices.

- **Hand Tracking:** Want to track a user's hand and finger movements? Lens Studio offers a **Hand Tracking** feature that can track the 3D position and rotation of the user's hands (and detect some gestures or poses). There are two approaches: **Hand Tracking 2D** (older, tracks hand presence and simple open/closed state in 2D) and **Hand Tracking 3D** (newer, full skeletal hand with joints for each finger). In Lens Studio 5, the **Object Tracking 3D** component can enable hand tracking mode or combined body+hand tracking [75] [76]. With full 3D hand tracking, you get joint transforms for the palm and each finger bone. This allows you to attach effects to the user's hands (like a glowing orb that the user "holds"), or trigger events when a certain hand gesture is made (e.g. fist vs open palm). Snap has templates demonstrating a hand-tracking-driven game (e.g. whack-a-mole using your hand) and interactions like pressing UI buttons in AR by reaching out. To use it, add a **Hand Tracking 3D** object from the Tracking section; it will similarly create a skeletal structure for a hand. You might use only one hand or track both – check the docs for how many hands can be tracked (usually one or two). Note: Hand tracking can be combined with body tracking – the Body Tracking 3D asset has an option "Track Hand" which if enabled will also track hands but if disabled, you can use a separate Hand Tracking component for possibly better focus on just hands [77]. As with body, test the performance. Hand tracking may not work on all devices (older phones might not support it fully or as smoothly).

- **Marker Tracking:** (discussed in Lens Types above) This technology lets the lens recognize a specific image in the environment. In Lens Studio, you add an **Image Tracking** object (which internally uses a **Marker Tracking component**). You then **provide a reference image** (the marker). At runtime, when that image is seen by the camera, the Marker Tracking component will move its object to align with the image in 3D. You can make it track a flat printed image or even a Snapcode. A typical usage is a portal lens activated by a poster: e.g. point camera at a movie poster, and the poster "comes to life" with animation on it. Marker tracking in Lens Studio requires a high-detail image (the docs recommend images with distinct features, not too repetitive or low-contrast [78]). You can import the marker via the **Marker Training** template or by adding the component and selecting your image file. It will generate a marker asset that the lens uses for detection. Keep markers fairly large (Snap suggests ~2048px image for quality) and note that extremely detailed images might be less robust at a distance (because of small feature tracking) [79]. Also, **Snapcode tracking**: you can set the marker to be a Snapcode so that when the user scans a special Snapcode, the lens overlays something on it (Snapcodes have encoded position info that the lens can parse). Marker tracking is a bit less common than face/world lenses, but very powerful for AR art installations or branded promotions involving logos/products.

- **World Tracking (6DoF SLAM):** This underpins all world lenses. Using ARCore/ARKit and Snap's own algorithms, Lens Studio can track the device's **position and orientation** in space, so that digital objects can stay anchored in the real world. In practice, you use the **Device Tracking** component set to **World** mode on your Camera (the default for new projects is often already world tracking if it's a world lens template) [17]. World mode means as you move your phone, the coordinate system of the scene matches real movement – objects appear fixed in your room. Snap's world tracking also provides **plane detection** (the Instant World feature attempts to hit-test surfaces quickly using the depth map before a full mesh is built [80] [81]) and a **Depth Texture** which gives a per-pixel depth estimate of the scene (even on devices without hardware depth sensor, Snap's **World Mesh 2.0** can approximate depth using AI – providing a dense depth cloud) [82] [83]. The **World Mesh** feature, when enabled, continuously scans and updates a 3D mesh of the environment – you can use this mesh for advanced effects like physics collisions with real world or occlusion (e.g. an AR object goes behind a real object that's been meshed). World Mesh 2.0 significantly improved quality and is supported on a wide range of devices (not just LiDAR iPhones) [82] [84]. In Lens Studio, using the mesh is as simple as toggling it on (Project Settings or adding a World Mesh asset) and then using a **Depth Occluder** material on an object to hide things behind geometry, or reading the depth texture in a custom shader. For simpler use, Snap provides a **Ground Grid** helper which finds a horizontal plane for you to place objects on the "floor."

In summary, for world tracking you will almost always: ensure your Camera has a Device Tracking (World) component, then place objects as children of the World object (so they move correctly). You might use the **World Object Controller** template/asset which lets users tap to place or drag objects in world (it handles touch-to-world coordinate mapping). This asset uses hit-tests against the depth texture to drop an object on the first detected surface [85] – which greatly improves UX (no need to wave phone around for 5 seconds; the object can appear immediately and then refine position when mesh is ready). If you are building a world lens from scratch, consider importing the *Instant World* template or the *World Object Manipulator* from the library, which gives two-finger rotate/scale gestures on a world object.

Tracking Scope & Multiple Targets: Lens Studio's **Tracking Scope** setting (under AR Tracking) allows you to control whether effects apply to the "Front Camera", "Back Camera", or both, and how multiple targets are

handled [86] [87] . For example, you can specify that a face effect should track **each face individually** or just the first face. Similarly, body tracking can track multiple people. Usually, the built-in components manage scope (e.g. a Face Mask component might have a "Face Index" property as mentioned). But be aware you can manage this in script if needed (iterating over face objects, etc.).

Finally, Snap has some special tracking modes: - **Landmarker tracking** for known world landmarks (like the Eiffel Tower) – Snap provides these to official creators for custom landmark AR, but these are more specialized and not broadly used in Lens Studio by casual creators. - **Location-based triggers** (Lens Studio **Location AR** and **Local Lenses**): if you need a lens to activate only at certain GPS locations or use Street Map data (Snap's "Places" API), that's also possible (under Lens Cloud/Location features). For example, city-specific lenses or persistent AR content at a location use these features. They are advanced and often require access to Snap's program (for persistent local lenses). But you *can* use the **Snap Places API** in a simpler way to get nearby locations data via Remote API (see section 13).

Takeaway: Choose the tracking tech needed for your lens. For a selfie filter – face tracking is your go-to (Lens Studio provides dozens of face effect presets so you rarely need to code the face math yourself). For putting objects in the environment – enable world tracking and optionally surface or mesh detection for realism. For full-body or hands – use body and hand tracking components and attach your content to the joints. Test these features on device because tracking quality can vary in different conditions (e.g. low light can affect AR tracking stability). And be mindful of performance: tracking multiple bodies and faces and hands **simultaneously** with physics and ML all at once might overwhelm some phones. It can be better to toggle certain trackers off when not needed (Lens Studio's **Tracking Scope** or enabling/disabling objects can help – e.g. disable body tracking if user switches to front camera for a face effect only).

Lens Studio's AR tracking is state-of-the-art, stemming from Snap's research (e.g. Snap acquired AI Factory for their body tracking). As a creator, you have a one-stop platform to track virtually anything: faces, bodies, hands, images, surfaces, and more.

7. Material Editor (Shaders & Materials)

To control how your objects look, you'll work with **Materials** in Lens Studio. A material defines the surface appearance (texture, color, shininess, transparency, etc.) and in advanced cases, can produce special visual effects. Lens Studio includes a **Material Editor** which is a node-based shader editor for creating custom materials without writing GLSL code [88] .

Material Basics: Each renderable object (2D or 3D) has a Material assigned. For example, a 3D mesh imported might come with a default PBR material; a Screen Image might use an Unlit Texture material. You can use built-in material presets (Face Mask retouch material, Distortion material, etc.) or create your own. Lens Studio supports **Physically-Based Rendering (PBR)** materials for 3D objects, which have channels like Albedo (diffuse texture), Normal map, Metallic, Roughness, Occlusion, etc., similar to other engines. These work with lights in the scene for realistic shading and reflections. For simpler needs, you might use unlit materials (which don't react to light, e.g. for UI icons or fullbright cartoon looks).

Using the Material Editor: The Material Editor is a workspace where you can add **Nodes** and connect them to define the material's shader. To create a custom material, in the Asset panel click **+ -> Material**, then open it. You'll see nodes like **Main Output** and you can add math nodes, texture samplers, colors, etc. This is very powerful: you can implement things like UV scroll (animated textures), fresnel glow, chroma keying,

procedural animations, and more. Snap's docs note that the Material Editor is "a new way to create your own custom materials *without writing any code*!" [88]. For instance, if you wanted a dissolve effect on an object, you could use a noise texture and compare it against a threshold, all in nodes, and feed that into opacity.

The Material Editor has various node categories (listed in the **Material Editor Nodes** reference [89] [90]): e.g. texture sampling nodes, math nodes (add, multiply), utility nodes (time, screen UV), PBR lighting nodes, etc. You can even create **Sub-Graphs** – reusable shader node groups for organization [91] [92]. There is also a special **Code Node** that lets you inject a bit of custom GLSL if you absolutely need something not provided by existing nodes [93] (use sparingly, as code nodes might be less cross-platform portable).

For many AR purposes, you may not need to build a material from scratch – Snap provides a **Material Library** (accessible via Asset Library) with pre-made materials: e.g. Glass, Hologram, Metallic paint, various VFX shaders. But knowing the Material Editor means you can customize these or create unique looks.

Common Material use-cases in Lenses:

- **Retexturing and Filters:** For example, apply a texture that was selected by the user onto a 3D object. Or adjust the hue/brightness of the camera feed in a post effect. The Material Editor can manipulate colors, UVs, etc.

- **2D Effects:** If you want a full-screen glitch or blurring effect, you often use a Post Effect material. These are shaders applied via an Orthographic full-screen quad or the PostEffect component on camera. You can design such a shader in Material Editor (e.g. sample the camera texture, warp it, and output). Lens Studio provides a **Screen Texture** node that gives you the camera image as an input to your shader, for doing filter effects [94].

- **Environment Mapping:** There are nodes for **Cubemaps** and reflection sampling. You can make shiny reflective materials by feeding in an environment map (Lens Studio includes a default environment cubemap, or you can use the **Environment Probe** component to capture the surroundings in real time for reflection). A PBR material with Metallic=1 will reflect the environment.

- **Animated Shaders:** Because lenses often benefit from dynamic effects, you can use time-based nodes. For example, the **Time** node gives an increasing time value; combine it with sine waves to oscillate values (like a pulsing emission). Or use a **UV Scroll** to make a texture move (common in effects like moving backgrounds, water flow, etc.). The Material Editor allows all that in a visual way.

- **VFX and Particle Materials:** If using the VFX Graph for particles, it also uses materials for how each particle looks. You might design a special material for your particles (e.g. additive glow, or particles that fade out).

The Material Editor is separate from the scripting system but you can link them via **Material parameters**. You can expose properties in your material (similar to script inputs) that can be adjusted at runtime via script. For example, you could have a "Progress" parameter in a dissolve shader, then in script gradually change that value to animate the dissolve. Lens Studio's API allows you to get a material and call methods

to set parameters (like `myMat.mainPass.someParameter = value`). Check the **Materials** section in the API for specifics.

For creators who are familiar with shaders: Lens Studio's Material Editor is essentially a visual shader composer that outputs optimized shaders for Snap's mobile runtime. If you have custom GLSL from Shadertoy, etc., you might integrate it with a Code Node, but often you can recreate it with provided nodes. There's also a **VFX Editor** which is node-based for particle systems – related but focused on particle behavior, whereas Material Editor is about pixel shading.

If you prefer coding shaders, you cannot directly write a full GLSL shader in Lens Studio (you'd use the Material Editor or code nodes). But since the Material Editor is quite comprehensive, you shouldn't feel limited. Snap actively updates it with new nodes (recent additions include noise functions, MatCap nodes, etc., and even a **Ray Tracing** feature for certain effects in latest versions [95]).

Practical workflow: Often you start with a base material type – e.g. "PBR" or "Unlit" – and then open it in the editor to tweak. If you just need to assign textures and colors, you can do that in the Inspector without diving into nodes. But for any fancy effect (glowing edges, holographic flicker, sketch outline), you click **Edit Material** and build the logic. Snap's official tutorials (Shaders 101) show examples like turning a photo into a comic book style via Material Editor nodes [96]. There are also community resources (many creators share material graphs on forums or the Lens Studio community site).

In conclusion, **Lens Studio's Material Editor gives you creative control over visuals**. It's a deep topic, but remember these points: - Use it when default materials don't achieve the look you need. - Keep materials optimized (mobile GPUs have limits – avoid extremely complex node graphs or large texture counts on many objects). - You can create **Subgraphs** to reuse effects and keep graphs clean [97]. - Check out the **Material Editor Tutorials** and the **Material Examples** in the docs [98], which step through building some interesting shader effects.

Materials can make your lens stand out – a unique shader can turn a generic 3D object into a mind-blowing effect. Don't be afraid to experiment in the editor; changes update live in the preview, so you can iterate quickly. And any material you create can be saved and reused in future projects, building your own library of cool shaders.

8. Audio in Lenses (Sound FX & Music)

Audio is a great way to enhance AR experiences – a sound effect or background music can make a lens more immersive. Lens Studio supports adding audio files and even integrating Snapchat's library of licensed music.

Importing Audio: You can import sound files (MP3 is recommended) by dragging them into the Asset Browser. Lens Studio will treat them as **Audio Track** assets [99]. It's suggested to keep files short (Snap recommends <15 seconds for size reasons) and mono if possible [100], since lenses have size constraints and many users view without headphones (mono ensures it's audible from the phone speakers clearly).

Playing Sounds with Audio Component: To play a sound in the lens, add an **Audio Component** to an object in the scene [101]. This component holds a reference to an Audio Track asset and has settings like

Auto Play, **Loop**, volume, etc. [102]. For example, if you want background music to start immediately, you can check **Autoplay Loop** on an Audio Component with a music track [103]. If instead you want to trigger a sound via script (e.g. play a "ding" when a game event happens), you leave autoplay off and call the `.play()` method on the Audio Component from your script [62]. In a script, you can get the component (e.g. via `//@input Component.AudioComponent myAudio`) and call `script.myAudio.play(1)` to play once (passing -1 would loop indefinitely) [62]. You can also stop or adjust volume via script.

Lens Studio allows **multiple sounds** to play simultaneously [104]. Common use-case: a background loop + short sound effects on events. Just be mindful of not layering too many because it could become noisy or hit performance issues decoding audio.

3D Spatial Audio: If you attach Audio to objects in the 3D scene, you can enable spatialization so the sound pan/volume depends on the object's position relative to the camera. The Audio Component has a **Spatial Audio** section with options: Distance roll-off, directivity, and stereo panning effect [105] [106]. To use spatial audio, you also need an **Audio Listener** component (usually on the Camera) [107] – in most cases Lens Studio automatically has the Camera act as listener. For example, if you place a virtual bird to the left of the user, you can have its chirping sound come more from the left speaker. Spatial audio is especially effective with headphones. If your lens is just playing UI clicks or a global music track, you can ignore spatial and leave sounds non-positional (the default is non-spatial unless you configure distance effects).

Snap's Licensed Music: Snapchat offers a catalog of music that can be used in lenses (similar to how you can add popular songs to Snaps). In Lens Studio, these appear as **"Licensed Music"** audio tracks. You don't import these from your computer; instead you **open Asset Library -> Music** section to browse available tracks [108]. Choose one and import it to your project. It becomes an Audio Track asset of type *FileLicensedSoundProvider* [109]. You play it in the lens like any other audio track (with Audio Component or Behavior). **However, licensed tracks have restrictions:** you can only use **one licensed music track per lens**, it will be mixed directly into the snap (so it records if user posts the snap with sound) [110], and you cannot combine it with other sounds playing at the same time [111]. Essentially, if you add a licensed song, that's the only audio that should play (Snap enforces that to avoid copyright issues). Also, if the track isn't available in the user's region or if device location can't be obtained, the music might be muted [112]. So as a best practice, either design your lens to work fine without the music (just in case), or provide alternate audio for regions where the song is unavailable. If a user applies a lens with licensed music, Snapchat will ensure proper attribution on the UI.

If you need sound effects alongside music, note the limitation that other simultaneous playback is *forbidden* with a licensed track [113]. A workaround is that Snapchat's system might allow short UI sounds (like taps) but not sure; generally, plan that you *either* have a music lens *or* a multi-sfx lens, not both. If you do include music, test carefully to see if your additional AudioComponents get muted.

Audio Effects and Beat Sync: Lens Studio provides **Audio Effects** that can be applied to an Audio Component (e.g. filters to change voice, reverb, etc.) [107]. These are more relevant if you record microphone input, which normal Lenses typically don't do unless you're doing voiceML or something. But one cool feature is **Beat Sync**: if you have a music track, Lens Studio can analyze its beat and tempo so you can synchronize visual effects to the music beat [107]. The **Audio Analyzer** or BeatExtractor can give you events or a parameter that oscillates with the music. For instance, you could make lights flash on the beat of the song. Snap's documentation has an **Audio Examples** section demonstrating beat synchronization. To use it, you add a **Beat Extractor** component, link it to an Audio Component, and then in script query the beat (or

use a Behavior: there is a Behavior trigger “On Beat”). This can add a lot of polish if your lens revolves around music.

Microphone input: By default, lenses do not use the microphone (except to capture user’s voice if they record a video – that’s out of lens control). But there are VoiceML features (section 13) where you might actively use mic input. If you do, you have to request microphone permission from the user (Lens Studio’s Voice ML or Audio Analyzer components do this). An **Audio Analyzer** component can take the mic input and give you frequency spectrum data or amplitude, which you could use for visualizers (like make an object scale up and down with the user’s voice volume) [114]. This is advanced but nice for music visualization lenses or karaoke style effects.

Implementation tips: Add your audio components to logical objects (for organization). E.g. have an object named “AudioController” with components for various sounds. Or attach certain sounds directly to the object that causes them (like a bouncing object carries its bounce sound). Use the **Logger** to debug if sounds fail to play (maybe the component isn’t found by script, etc.). Lens Studio logs might also warn if you violate audio rules (like trying to play two music tracks).

Also note: **User Experience** – many Snapchat users have sound off by default. It’s good to ensure your lens still works without sound (Snap will show a small sound icon suggesting to turn sound on if the lens has audio). Typically, add an on-screen hint like “🔊” icon if your lens is much better with sound, to encourage them to unmute. Keep audio volumes reasonable (don’t blast at max volume, as it could be jarring).

Performance and Size: Sounds add to lens package size, but MP3 is quite compressed. Follow best practices: trim audio to only what you need (don’t include a 3-minute song if you expect average user to use lens for 10 seconds; maybe cut a 15–30s highlight loop). And use mono to half the data size (stereo not critical unless you have complex spatial stuff).

To summarize, **Lens Studio audio workflow**: 1. **Import or select audio asset** – your own MP3 or a Snapchat-provided track. 2. **Add Audio Component** to scene – reference the audio asset, set autoplay/loop if needed. 3. **Trigger playback** – either automatically via autoplay, via a **Behavior** (e.g. Behavior: Play Sound On Mouth Open), or via script (`audioComponent.play()`). 4. **Spatialize or mix** as needed – set spatial settings if the sound should behave 3D, or leave default for UI/global sounds. 5. **Test** on device – ensure volume and sync are correct. Check that it records into the Snap if desired (for music, enable “Mix to Snap” on the Audio Component so that when user sends the snap, the music is included – otherwise by default, user’s microphone audio would overlay) [115]. *Mix to Snap* basically means use this audio in the output video; you usually want that for lens-intended sounds (especially music), otherwise the recorded snap might be silent or just have ambient noise.

Adding audio can really increase engagement – even just a satisfying “click” or “pop” sound on an interaction, or a background ambiance for mood. So definitely leverage it. Lens Studio makes it straightforward with the Audio Component and script control. Just remember the **one licensed music track limit** if you go that route, and always respect Snapchat’s community guidelines for sound (no overly loud or startling sounds that would create a bad experience).

9. Interactivity: Touch Input, Gestures, and Physics

AR experiences shine when users can **interact** with them naturally. Snapchat lenses primarily use **touchscreen input** (taps, drags, pinches) and sometimes device motion as forms of interaction. Lens Studio provides both high-level and low-level ways to handle input.

Touch Input Events: The simplest way to respond to user touch is via **Touch Events** that Lens Studio exposes. These are global events that fire when the user touches anywhere on the screen: **TouchStart**, **TouchMove**, **TouchEnd**, and **Tap** (a quick tap) [116]. You can add a script and create events for these. For example, a common pattern is: *on tap, do something*. In a script you might do `script.createEvent("TapEvent").bind(function(eventData){ ... });` to execute code on tap. Or use the Behavior script: "Touch Event: Tap -> do X". These events don't tell you where on the screen the touch happened by default, they're just global triggers (for location-specific interaction, we use the Interaction component, discussed next). But for many cases, you just need to know *that* a tap occurred – e.g. to trigger an animation or cycle to the next effect.

Lens Studio note: if you don't override it, some gestures are handled by Snapchat app itself (e.g. double tap flips camera, long press activates scan). If your lens uses those gestures, you should **enable Touch Blocking** to prevent Snapchat's default action [117]. You can enable touch blocking via a small script or Behavior (there's a Behavior response "Block Touch"). If you set **Touch Blocking = true** on an Interaction component or via script for the whole lens, then user input will only affect your lens and not the app UI (so double-tap can be used by your lens logic, etc.). Use this carefully – users are used to double-tap to flip camera, so if you block it, maybe implement an alternate camera flip button or instruct them.

Object-Specific Touch – Interaction Component: Often you want the user to tap *on a specific object* (like tap a 3D model to animate it). For this, Lens Studio has the **Interaction Component** [118]. You add an Interaction component to any scene object that has a visual (either 3D mesh or 2D UI element) [119]. This component essentially makes that object "touchable" and can detect touches within its bounds. It has properties: you can restrict which camera it uses to determine hits (usually leave default so it picks the right active camera), and you assign which Mesh or UI element to use for hit test (by default, it auto-uses the object's own MeshRenderer or ScreenTransform) [120] [121]. Once an object has an Interaction component, you can handle events on it via script: the Interaction component offers its own event callbacks, slightly different from the global ones. You don't use `createEvent("TapEvent")` for these; instead, you can register through the component API or (easier) use the **Inspector**. In the Inspector, the Interaction component will list events like **"Touch Start"**, **"Touch End"**, **"Tapped"** specifically for that object. You can drag a Script onto the object and in the script's properties tie functions to those Interaction events (Lens Studio's docs show that you can do `script.createEvent("TouchStartEvent").bind(...);` but to specifically get the Interaction component events, you use `scriptInteractionComponent.onTouchStart` or similar – however, LS now auto-integrates them if you bind in Inspector). The documentation suggests that **using Interaction Component events is the preferred way** for object-specific interactions [122].

So the flow: Suppose you have a 3D present box object and you want it to open when tapped. You'd add Interaction Component to it. Then maybe add a script on it with a function `function onTap() { /* play animation */ }`. In the Inspector, link the Interaction's Tap event to call `onTap`. Done. Now when user actually taps on that 3D model (and not elsewhere), it triggers that

function. This is more precise than global tap (which would trigger even if they tapped some other part of screen).

The Interaction component handles both **2D and 3D** objects – e.g. for a Screen Image button, adding Interaction will let it detect taps within its rect [123]. It uses the object's bounding box for hit-testing, so be aware of that (for 3D, it's axis-aligned bounds in world; for 2D, it's the rect bounds). If you need more fine hit test (like tapping on a complicated mesh shape), you can approximate by attaching multiple smaller colliders or use a mesh's polygon, but usually bounding box is enough for UI.

Multi-touch gestures: Lens Studio doesn't have built-in "PinchEvent" or "Pan with two fingers" event by name, but you can implement these by tracking TouchMove and looking at positions. However, Snap provided a handy **"Manipulate"** asset (the World Object Manipulator template) which already implements pinch-to-scale and twist-to-rotate for an object. If you want to allow the user to **scale/rotate/drag** an object with gestures, it's easiest to import that template. Under the hood, it uses an Interaction component to track touches and some math to determine gesture intent (distance between two touches for zoom, etc.).

Basic custom approach for pinch-zoom: use TouchStart/Move events to get positions (`event.getTouchPosition()` gives normalized screen coords for a touch event in script). If two touches are active, calculate their distance and compare to prior distance to determine scale change. Similarly angle for rotation. It's a bit of work, so again, the community often reuses existing scripts for this.

Gesture triggers via ML: On a different note, Snap's **Hand Tracking** can detect certain gestures (like open hand vs closed fist) which could be considered "gestures" beyond touch. If your lens needs that (e.g. "make a fist to change something"), you'd rely on the Hand tracking component events or script queries (like `HandTracking.getHand(0).isFist`). This goes beyond normal touch but is an interactive gesture. Snap's **Body tracking** can also detect things like jumping jacks or raised arms if you program it. And face gestures (like raise eyebrows to trigger) are accessible via face expression events (e.g. Behavior: "Brows Raised" trigger). All these allow interaction using the user's body, not just touch – something unique to AR!

Physics-based Interactions: Lens Studio has a built-in **Physics Engine** that can simulate real-world physics for 3D objects [124]. This means you can make objects have gravity, bounce off each other, be thrown by the user's motion, etc. The physics is powered by projectiles and colliders – you add components like **Rigid Body** (with properties: dynamic, static, mass, bounciness) and **Collider** (box, sphere, capsule shape) to your objects, and Lens Studio will simulate them. You can then respond to collisions or apply forces via script. For example, a lens game where you tap to spawn a ball and it falls and ricochets around can be done with physics components rather than manually animating everything. The **Physics simulation** runs each frame and can dramatically increase interactivity – things feel more real when they react to gravity or user input realistically.

To use physics: enable physics in Project Settings if needed (Lens Studio physics should be on by default in LS 5). Add a **RigidBody** to any object you want to move physically, and a **Collider** shape to both that object and any object it should collide with (including the "ground" – you might use an invisible ground plane with a collider). The engine then does the rest. You can adjust global gravity (default is -980 cm/s^2 in Lens units, which are centimeters, meaning roughly Earth gravity) [125]. Through script, you can call methods like `rigidbody.addForce(new vec3(x,y,z))` to throw objects.

One fun feature: you can tie physics to **user gestures** – e.g. have the user drag an object and then release to "throw" it. You'd on touch-drag, move the object (perhaps turning off physics while dragging), then on release, turn physics back on and set the rigidbody's velocity based on the drag direction/speed to simulate a throw. The physics engine will handle the rest (ballistic trajectory, bounce).

Lens Studio's physics also supports **Triggers** (detect overlap without collision) and you can respond via script events. The **Behavior helper** script even includes a "Collision Enter/Exit" trigger [126], so without code you could make something happen when two objects collide (like particle burst on hit). Physics is great for creating mini-games in AR or playful interactions like a bunch of objects that topple when you tap one.

Do note physics can be heavy on CPU if overused (multiple complex shapes). Simpler shapes (capsules, spheres) perform best. Also remember mobile limitations: no super dense meshes as colliders and no thousands of rigid bodies (keep it maybe under a few dozen active bodies at once for performance).

Putting it together: An interactive lens often combines **touch + physics**. For example, consider a bowling game lens: - The user **drags** (touch) a bowling ball to aim. - On release (tap end), you **apply a force** to the ball's rigidbody to roll it forward. - The pins are rigidbodies that react (fall) on collision. - Maybe the user can also swipe on screen to apply spin (touch move influencing ball's velocity direction). This synergy of input and physics yields a fun, intuitive game – all achievable in Lens Studio.

Another example: a simple interactive face lens – you have an emoji floating by your head; if you **open your mouth** (face event), the emoji "falls" into your mouth (enable gravity on it), and maybe you **tap** to spit it out (apply upward force on tap).

It's good to leverage the **Behavior Script** for quick interactions. It covers many triggers: mouth open, smile, eye blink, tap, etc. and responses: play animation, enable object, play sound, etc. Without writing a line of code, you can implement "When user raises eyebrows, switch to next effect" or "When user does a thumbs-up (if you have Hand tracking recognizing thumbs-up), show a thumbs-up graphic". The Behavior script is essentially a collection of common interactive patterns baked in by Snap.

Finally, always test the interactions on a device. Touch precision, multi-touch, and performance of physics can differ on device versus PC preview. Use Snapchat's pairing to live-test and check that taps register as expected (sometimes screen protectors or device differences can affect sensitivity). Check that UI elements sized appropriately for fingers, etc.

Physics optimization: On weaker phones, too many physics objects cause low FPS. Lens Studio's Physics Overview mentions limitations (like avoid too small or too large colliders, and watch out for tunneling issues if objects move too fast) [127] [128]. Stick to reasonable object scales (Lens units default to centimeters, so gravity is set expecting models are in that scale – e.g. 100 units ~ 1 meter). If your model is huge or tiny, tune the physics or scale accordingly.

In summary, **interactivity** in Lens Studio can be approached through: - **Touch-based UI** (taps, swipes) via Interaction components or global events. - **Gesture and behavior triggers** using face expressions, body movements, and helper scripts. - **Physics interactions** to add realism and dynamic response to user input and gravity. - **Scripting** to glue these together when needed for custom logic (e.g. complex gesture recognition, combining inputs, etc.).

This transforms your lens from a static filter to an engaging experience where the user is an active participant, not just a viewer.

10. Templates and Sample Projects

Don't reinvent the wheel – Lens Studio comes with a wealth of **templates, examples, and preset projects** that you can use as a starting point. Especially when learning, templates are your best friend to understand how certain effects are built.

Built-in Templates (Starter Projects): When you open Lens Studio's Home Screen, you'll notice a section for **Starter Templates** under different categories (Face, World, Music, Games, etc.). These templates are essentially example Lens projects that demonstrate a specific feature or style [129]. For instance: - "Face Paint" template: shows a basic face mesh painting. - "Hair Color" template: uses hair segmentation to recolor hair. - "Makeup" or "Retouch" templates: how to apply beautification filters. - "3D Body" template: multi-person full-body tracking with an avatar. - "Portal" template: sets up a world portal (with doorway and environment inside). - "Landmarker" templates: for famous landmarks (if available to you). - "Game" templates: simple AR games like a whack-a-mole or a basketball toss. - "Voice and Hand Gesture" templates: maybe one that responds to a "high-five" gesture or voice command (if available in assets).

When you create a new project from a template, Lens Studio basically pre-populates the scene, assets, and scripts used for that example. You can then **customize it** with your own assets and tweaks. This is a fantastic way to learn best practices because these templates are made by Snap's team or verified creators, so they usually follow optimal setup.

For example, the **"World Object Manipulator"** template includes an object you can place and interactions for pinch/rotate – you can drop your own model into that and instantly have a working drag/drop object in AR. The **"Upper Body Interaction"** template might show how to attach objects to shoulders or hands and use simple triggers. The **"ML Object Detection"** template (if available) could show using a SnapML model to detect an object and spawn content.

Asset Library Examples: In the **Asset Library** (accessible via the toolbar or Asset panel), there's a category for **"Lens Examples"** or **"Components"**. These are like mini-templates: you import them into an existing project. The Snap docs list various Lens Examples like *3D Text*, *Cutout*, *Freeze Frame*, *Portal*, *Quiz*, etc. You can import these as **Asset Packages**. When you import, say, the **"Quiz"** example [130], it will bring in a prefab and scripts that implement a quiz game (likely with question UI, randomizer). You then drag the prefab into your scene, and perhaps just replace the question content. These examples are slightly different from templates in that they can be added to your project piecemeal as features. Another example: the **"Tween" example** [131] likely imports the Tween Manager and a sample usage.

A particularly useful category is **"Components"** (in Examples or Asset Library) – these are ready-made custom components that add functionality. For instance, Snap has a **"Physics Toy"** component that, when added, might do a quick physics demo, or a **"Leaderboard"** component for games (connected to Snap's backend for scoring) [132] [133]. The Asset Library has things like **"UI Kit"** (pre-styled UI widgets for Snap's design, e.g. buttons, toggles) and **"Avatar Gestures"**, etc. Using these not only saves time but ensures you use features correctly.

Best Practices for Customizing Templates:

- **Read any Guide or Description:** Many templates come with a text guide (often in the Lens Studio docs or as comments in scripts). For example, if you open the template, check the Script Components or objects for notes on what to replace. The *3D Text* example in docs explains each sub-example and how to use it [134] [135].

- **Swap Assets**: The primary way to customize a template is to replace the template's example assets with your own. E.g. if template has a placeholder 3D model, import your model and assign it in place (be mindful of hierarchy and component settings). If it has textures, replace them with your art. For face lenses, many templates have a **placeholder texture** (like a face paint image) – just double-click it and import your texture or edit it in an external editor.

- **Adjust Parameters:** Templates often expose custom script inputs or materials you can tweak. For example, a template might have a "Customization" object with a bunch of properties you can set in Inspector (like color of effect, speed of animation, number of spawned objects). Play with these to see effect changes.

- **Unpack Prefabs if needed:** Some imported examples come as a Prefab (as noted earlier). If you need to modify internal parts of it (like the script inside, or the hierarchy), right-click and **Unpack** it [41]. This will break the link and let you edit pieces. After unpacking, you can delete or reconfigure parts without worrying that it's a linked asset. For instance, the Quiz prefab might have 3 example questions; you'd unpack and then edit the text for those questions or add more.

- **Maintain Structure:** Try not to remove core objects or components that the logic relies on unless you know what you're doing. Templates are carefully set up; removing something could break a script reference. If you don't want a feature, it can be safer to **disable** it (checkbox in Object Inspector) rather than delete, at least initially.

- **Follow the Template's Patterns:** If you want to extend it (like add a new level to a game template), imitate how the template does it. If the template uses a script list of objects, add to that list rather than making a completely separate mechanism. This keeps things consistent.

Snap also highlights certain **"Tags of the Month"** on the home screen – these indicate popular trends (e.g. "NewYear2026" or "CartoonStyle"). If you make a lens matching those and tag it when publishing, it might get featured. Templates often correspond to trending effects (like year-end frames or holiday themes). Keep an eye on those if you want inspiration that aligns with what's popular.

Learning from Templates: Even if you intend to build from scratch, it's illuminating to open relevant templates and see how Snap's team set it up. For example, if you plan to use **Segmentation**, open the "Segmentation" template [136] – you'll see how they structure the cameras (they likely have two cameras: one rendering background, one rendering foreground subject, using the mask). This can teach you the correct layering technique. Or to use **SnapML**, open a SnapML example to see how to add the ML component and how to feed its output to something visual.

Community Templates: Beyond official ones, the Lens Studio community (forums, Lens Studio Discord, Lens Studio subreddit) shares templates and assets. Sites like Lenslist sometimes feature project

breakdowns. One highly recommended resource is the "Official Lens Studio Examples" page [137] (we saw a partial list) – use those because they're tested.

Template Limitations: Sometimes templates use specific assets that you shouldn't reuse for publishing (like a Snap-branded asset might be for demo only). Replace all demo content with your own or license-free content. Also check script compatibility if you combine multiple templates – there could be conflicts (e.g. both define a `global` variable with same name). Test thoroughly if merging.

Projects and My Lenses: If you start from a template and create a cool lens, you can save it and publish like any other. It's *your* lens now. Snap doesn't restrict using templates – they're provided to encourage creation. Just ensure you change enough so it's original (many people may publish the basic template if they don't change it, leading to many duplicates; a more unique spin will stand out).

Quick List of Notable Templates/Examples to explore: - **Face**: Face Inset (cut out parts of face), Face Stretch, Big Mouth, etc. - **World**: Pet AR (puts a virtual pet on floor), Plant Grow (spawn a tree in world). - **Games**: Camera Game template where you move to control or use head pose to steer something. - **ML**: "ML Object Classification" (identifies objects from camera and shows label), "Baby Yoda ML Example" (just a fun model example). - **Try-On**: Shoe Try-on template (foot tracking), Hat Try-on (head tracking). - **Multiuser**: Connected Lenses template (if available to you, an advanced template showing multiuser AR setup). - **Lens Cloud**: if using Cloud features, templates for Leaderboard, or Location-based lens (like AR treasure hunt using geolocation).

Using templates, you **accelerate development** and also learn proper implementation patterns. Each template is like a mini-tutorial baked into a project. Don't hesitate to dissect them – move one step in a template and see what components it has, how the hierarchy is arranged, etc. Over time you'll internalize these patterns and might not need templates, but even pros often start from one to save time.

11. Optimization and Performance Tuning

Creating a lens is one side of the coin – making sure it **runs smoothly and fits within Snap's size limits** is equally important. Mobile AR is demanding, so you should always keep performance in mind. Snapchat also enforces certain limits (lens file size, memory usage) especially for official or sponsored lenses.

Lens Size Limits: Currently, a submitted lens must be **under 8 MB** in compressed size [138]. (In the past it was 4 MB for community lenses; Snap increased it to 8 MB, but still recommends staying much smaller for better user experience.) In Project Info you can see your lens's size; Snap recommends aiming for **2 MB or less** for fast loading [139]. Why? Because lenses are often opened spontaneously – if it's too large, users might skip it if it takes long to load. For sponsored (paid ad) lenses, Snap actually still advises <4 MB for the broadest reach [140] (older low-end devices and to ensure quick load when served as an ad). Also, if using SnapML, you get an extra budget up to 10 MB for the model files [141] (that doesn't count toward the 8 MB lens size). But only use that if absolutely required for ML features.

Optimizing Assets: The biggest contributors to size are **textures**, **models**, and **audio/video**. Lens Studio has automatic **compression for textures** [142] – by default it chooses a good balance for performance (compress to GPU-friendly formats). In the Inspector for each image, you can choose compression level (e.g. Optimize for Performance vs Size) [143]. If your lens is bumping against size limit, try setting some large

textures to “Optimize for Size” which trades a bit of quality for smaller footprint [144]. Also scale down resolution of images if they’re larger than needed – e.g. a 2048px texture for something that appears small could be downsized to 1024px with little visible difference and quarter the data size. **Use JPG for photographic textures** (lossy but small) and **PNG for graphics that need alpha** (and let LS compress them). For sequences or GIF-like animations, consider using **Animated Texture** feature with good compression or even Snap’s new **Compressed Texture sequences**.

For 3D models, reduce polygon count and remove unused rig bones or blendshapes. Convert heavy models to **.gltf** or **.obj** if possible (they tend to be more compact than FBX in some cases). If a model has high-detail textures or lots of them, see if you can use a single material or lower resolution. Also **disable unused parts** – e.g. if your FBX includes multiple meshes but you only need one, delete the rest in a 3D editor before importing or in Lens Studio if possible.

Audio optimization: Use shorter clips and MP3 compression. Mono instead of stereo (half the size) [145]. Lower the bitrate if high quality isn’t needed (e.g. voice lines can be lower bitrate than music).

Lens Studio’s **Project Info** (Project Settings) panel shows the current lens size and a breakdown by asset. Use that to identify big assets and target them.

Memory and Performance (FPS): Snap expects lenses to maintain a good framerate (target 30 FPS). There is a performance overlay (when paired to device, tap the bug icon) that shows FPS, memory, etc [146] [147]. Guidelines: **keep FPS >= 15** at least, preferably 30 on most devices [148]. Memory should stay under ~150 MB usage [148]. The “LAT” (Lens Activation Time) – the time it takes to initially activate – should be low (for ads, must be <650 ms on benchmark device) [149]. LAT is influenced by how much initialization your lens does (e.g. huge models or many scripts can delay startup). Use the automated tools: if you upload a lens draft to My Lenses, Snap runs tests and will warn if performance is poor.

Optimization Techniques:

- **Reduce Draw Calls:** Each object with a different material or each Screen Image may add a draw call. Merging meshes and using as few materials as possible helps. Atlas textures if you can (combine multiple small textures into one sheet and use UV offsets, so multiple objects use one atlas = one material). For 2D elements, use a single Image component with sub-region instead of many separate images.

- **Level of Detail / Culling:** Lens Studio doesn’t have built-in LOD switching, but you can manually hide objects that are not in view or too far. For example, disable certain effects when not used or when off-camera. Use **Render Layers** and layer culling if needed to limit what each camera renders (to avoid overdraw).

- **Particle Limits:** If using the VFX particle system, be mindful of max particles, spawn rate, and complexity of VFX graph. A few hundred particles are okay; thousands might drop FPS on mobile. Also, particles with lots of overdraw (semi-transparent large sprites) can hurt fill rate – optimize by using smaller or fewer particles or enabling **Depth Write** on particles that don’t need layering so they don’t overdraw as much.

- **Script Efficiency:** Avoid heavy computations in Update events. Do not allocate lots of objects every frame in JS (garbage collection can cause hitches). If possible, use built-in components instead of scripting equivalent behavior, as native code is faster (e.g. use Behavior script or tween component rather than manual tween logic in JS every frame). If you do have an expensive operation, consider if it can run once or at intervals rather than every frame. Also note that TypeScript won't inherently run faster than JS; optimization is about algorithm choice and doing less work overall.

- **SnapML Performance:** If you include an ML model, be aware of its complexity. Models with many millions of parameters may slow down lens or consume memory. Try to use models optimized for mobile (pruned, quantized if possible). Snap's documentation on SnapML suggests keeping model size and layer counts reasonable. Always test on a mid-range Android device if using ML, since that's often the slowest case.

- **Profiling Tools:** Lens Studio has a **Lens Performance Profiler** which can capture a trace of CPU/GPU usage per frame [150]. This is advanced but if you suspect a bottleneck, you can use it to see where time is spent (rendering vs script vs physics, etc.). Snap also groups phones by tiers ("Clustering Phones") [151] [152]. Ensure your lens runs on lower tier devices by either testing or reviewing Snap's automated test results. If the results show poor performance on low-end, consider scaling down effects for those devices (Lens Studio has an API to detect device performance class, so you can script something like "if low-end device, disable complex particle system").

- **Memory:** 3D models and textures use runtime memory. Keep textures as small as acceptable. If you have large models/animations, ensure they are necessary. Remove any assets in the project that you ended up not using – they still add to lens size and possibly memory if loaded. The **"Lens Hints"** panel in Project Settings might give suggestions (like "Asset X is not referenced" or "Too many materials on one mesh").

- **Testing edge cases:** If your lens can potentially spawn many objects (e.g. user can tap to add stickers endlessly), consider putting a limit or recycling objects to avoid running out of memory or crashing. Lenses are short-lived usually, but a user could theoretically keep a lens open a long time, so guard against unbounded memory growth (like don't leak by constantly instantiating new objects without removing old ones).

- **Quality vs. performance trade-offs:** Sometimes you can sacrifice a bit of quality to boost performance. Example: use **unlit materials** instead of PBR if lighting isn't crucial; this saves the cost of lighting calculations. Or use a **lower resolution** device camera texture (Lens Studio allows you to set preview resolution, but in real use it's device camera feed – some lenses use a half-res render target to apply heavy post effects then upscale, to lighten GPU load). Also consider turning off **"High Complexity" features** you may not need – e.g. if your lens doesn't need 60fps tracking for body, you might lower tracking frequency (Snap doesn't expose that, but for ML you can process every 2nd frame for heavy models).

Snap's docs and video tutorials highlight **Top 5 Optimization Tips** [153]: 1. Reduce lens size (faster load). 2. Simplify materials and meshes (reduce draw calls). 3. Use appropriate texture compression. 4. Profile on device with performance overlay (FPS, etc.) [154] [147]. 5. Test on various devices (especially if you have access to both high-end and low-end Android).

Remember, a smooth lens at 25-30 FPS with a bit lower fidelity is better for user experience than a gorgeous lens that lags at 5-10 FPS. Users generally tolerate simpler visuals more than choppy performance.

Finally, **Snap's submission process will flag major issues**. If you submit a lens that crashes devices or consistently drops below 10 FPS, it might be rejected or demoted. So it's in your interest to optimize. Use the **"Optimizing Your Lens"** guide [155] [156] and checklist in Lens Studio as you finalize your project.

12. Publishing: Testing, Submitting, and Managing Lenses

Once your lens is built and optimized, you'll want to **publish it to Snapchat** so others (or just you) can use it. Snapchat provides the **Lens Studio application** and the **Snapchat app** for testing, and the **Snap "My Lenses" dashboard** for submission and management.

Testing on Device: We covered pairing your device earlier – it's crucial to test the lens on at least one real phone before publishing. Lens Studio's Preview gives an approximation, but the device test will reveal actual performance, camera feed quirks, and overall feel. To test, use the **Preview Lens (Send to Snapchat)** button in Lens Studio to push it to your paired Snapchat [157] [158]. The lens will appear in your Snapchat carousel as an unlocked lens (only visible to you) [159]. Try all interactions, switch front/rear camera as needed, record a snap to ensure everything (including audio if any) records properly. Check the lens under various conditions: different lighting (for tracking), different faces or objects if relevant, etc. If you paired multiple devices (you can pair more than one phone) [160], test on both iOS and Android if possible, or high-end vs low-end.

Final Project Settings: Before submission, fill out the **Project Info** (Project Settings) in Lens Studio. Give your lens a clear **Name** (max 18 characters, only letters/numbers/spaces) [161] – this is what users see when they unlock it. Choose appropriate **Lens Icon** – an eye-catching icon helps in the carousel. Lens Studio allows generating an icon (with the GenAI Suite if you want) or importing an image [162] [163]. Ensure it represents the lens content clearly (Snap suggests simple, cartoon-like, bright icons, 320x320 PNG with their template frame) [164]. Also attach a **Lens Preview** video – a 5-10 second demo video of your lens in action – which shows up in Lens Explorer. You can capture this via Lens Studio's preview or upload one [165]. Previews are important for discovery; choose a good showcase (for example, if it's a transformation lens, show before/after, etc.).

Also in Project Settings, set **Lens Works On**: front camera, rear, or both [166]. If your lens only works properly on one side, specify that (e.g. a face lens wouldn't make sense on rear, so disable rear camera). If you built a Spectacles-specific lens or a web one (Camera Kit), set those toggles appropriately [167], though most public lenses are just for mobile Snapchat.

Submission via My Lenses: When ready, click **Publish Lens** (the cloud icon in Lens Studio or the Project Settings publish button). This will prompt login to your Snap account and take you through the submission flow (alternatively, you can go to the My Lenses dashboard on web). You'll need to provide: - **Lens Name** (auto-filled if you did Project Settings). - **Lens Icon** (upload if not done). - **Lens Preview** video (upload .mp4 or choose from default examples, but best to have your own). - **Categories/Tags:** You can pick some tags and categories that describe your lens (e.g. "Gaming", "Funny", "World" etc.). Snap uses these for discovery, and trending Tags of the Month if relevant [168]. - **Description (optional):** A short blurb about your lens. This currently is only visible in Lens Explorer details, but it can help give context or instructions if needed ("Open your mouth to...", etc.). Keep it concise and clear. - **Visibility:** You can choose to publish as **Public**

(discoverable) or **Hidden/Private**. Public means anyone can find it via search or explore if it gains traction. Hidden means only people with the direct link or snapcode can use it (it won't show up in search). Hidden is useful for test lenses or personal ones you don't want broadly available. - **Region Lock (if any):** Generally you'd make it available everywhere. But you can geo-lock if for some reason it's region-specific (business lens targeting one country). - **Age filter:** Lenses must be appropriate for 13+, and some might be flagged as 18+ (like those involving alcohol references). Snap's submission will ask if it's appropriate for all ages or needs a 18+ gate (this can affect who sees it).

After filling these, you submit. Snap's system will then **review your lens**. Community lenses typically get reviewed and approved fairly quickly (often within minutes to an hour), checking for policy compliance (no prohibited content, etc.). If something is wrong, you might get a rejection email with reason (e.g. "lens contains copyright content" or "lens doesn't function as intended"). You can fix and resubmit if that happens.

Once approved, your lens becomes live. You'll get a **Snapcode** and a **Lens Link** (swipe up link). These are in the My Lenses dashboard. You can share the Lens Link (it's a URL that, when clicked on mobile, opens Snapchat and the lens). The Snapcode image can be shared anywhere for people to scan in Snapchat.

Your lens will also be accessible via **Search in Snapchat** typically by its name and maybe keywords (e.g. if someone searches "puppy ears", and your lens has those, it might appear depending on popularity).

Managing Lenses: In the My Lenses portal, you can see all lenses you've published. You can **update a lens** by uploading a new version (for bug fixes or improvements). Be mindful, updating replaces the lens for everyone, and might reset its popularity metrics. But if it's a fix or enhancement, do update. If you drastically change what it does, consider making a new lens instead to not confuse existing users.

You can also **view analytics** for your lenses on that dashboard: views, plays, shares, etc. It's gratifying and insightful to see how many people use your creation. If a lens goes viral, you'll see big numbers here.

If you want to **remove** a lens, you can mark it as inactive in the dashboard. You can also toggle whether it's discoverable. If you want to collaborate or transfer a lens to someone (for client work), Snap allows transferring ownership via the business manager if needed.

Lens Explorer and Discovery: If your lens is public and gets some traction (or Snap's team likes it), it might appear in the **Lens Explorer** under relevant categories or trending sections. Also Snapchat's camera can surface lenses contextually (e.g. if user says "Hi" to Camera Scan, it might suggest lenses that say hello). Snap's discovery algorithm is somewhat opaque, but usage, quality, and relevancy tags all help.

Promo Tips: To get your lens out there, share it on social media (with the Snapcode or link). There are communities for Snapchat lenses (subreddits, etc.) where you can post your new lens. Also, a good preview video helps if you share on TikTok/Twitter to attract people to try it. If you made something unique or aligned with a trending meme, it might naturally pick up usage.

Lens Creator Rewards: As of early 2026, Snap has a program to reward top performing community lenses with cash [169]. If your lens gets extremely popular (measured by views in a month), Snap might pay you through the Creator Rewards program [170]. This is not something you apply for; Snap will notify you if you qualify. But it's an extra incentive to make quality, engaging lenses.

Sponsored Lenses: If you are working on a brand/sponsored lens, the publishing goes through Snapchat's business channels, often with additional requirements (like attaching it to a campaign). For personal or community lenses, you just self-publish via Lens Studio as described.

Testing and QA: Before final publish, consider a round of QA: - Test in different lighting (dark room vs bright sun) – tracking or AR might behave differently. - Test with different people if face lens (male/female, various skin tones, with/without glasses, etc.) – ensures your effect works for everyone (Snap encourages inclusive lenses that work regardless of face shape, skin tone, etc. – e.g. avoid face paint colors that don't show on all skin tones). - If possible, test on both a high-end phone (e.g. latest iPhone) and a low-end Android (3-4 year old mid-tier). If you notice lag on the low-end, you might implement a simplified mode (maybe disabling some effects via `systemInfo.performanceRating` check). - Check memory if possible (the bug icon stats). If your lens memory (RAM) is climbing over 100MB, see if you can optimize assets.

After Publishing – Updates: User feedback might come via Snapchat itself (people can rate lenses or leave comments in Lens Explorer). Pay attention if users report something like "lens doesn't work on X phone" or "it crashes when I do Y". You can then debug and update the lens. The nice part is updates propagate immediately (no need for users to re-scan code; next time they load lens it's the new version).

Lens Lifecycle: Public lenses remain available as long as you keep them active. If you want to retire one, you can deactivate it. Also note Snap may remove lenses that violate policies or if they become outdated with app changes. Keep an eye on any Snap emails about your lens (e.g. if a music license in your lens expired, Snap might notify you to update the track or they might deactivate it).

Publishing a lens to Snapchat is relatively straightforward compared to app store submissions – it's free, quick, and you can iterate often. So don't be afraid to publish and then improve. Some creators publish early to secure a lens name or concept, then refine with updates as they gather feedback.

13. Advanced Features and Integrations

As you become proficient, you may tap into Lens Studio's more **advanced features** to create truly innovative experiences. Some notable advanced capabilities include machine learning (SnapML), external data/API integration, Connected Lenses (multi-user AR), Lens Cloud services, etc. We'll focus on a few mentioned: **ML features, external APIs, and data sources**.

SnapML – Machine Learning in Lenses: SnapML allows you to import custom neural network models (in ONNX format) into Lens Studio [171] to power effects like object detection, segmentation, translation, etc. This is a game-changer feature – you can train a model (using Python/PyTorch/TensorFlow and export to ONNX) to recognize certain objects or perform some image transformation, and run it on the user's device in real-time! Examples of SnapML use-cases: - **Custom Segmentation:** Beyond the built-in person/hair/sky masks, you could train a model to segment *anything* (e.g. cats, food, cars). Snap provides a "Custom Segmentation" template that shows how to use an ML model to get a mask and then apply it [172]. - **Object Detection/Classification:** You can have a model that detects objects in the camera view (e.g. identify the type of plant, or find a logo). Then your lens can respond (spawn AR info labels on detected objects, etc.). For instance, an ML model could detect a dog breed from the camera – the lens then displays the breed name above the dog. - **Style Transfer / Filters:** ML models can be used to apply a certain art style to the camera feed or do AI effects. Snap had community examples like turning the scene into a painting in Van

Gogh style via an ML model. - **Facial or Pose-driven ML:** If built-in face expressions are not enough, you could use an ML model on the face image to recognize a specific emotion or trait. Similarly, Snap's **Landmark tracking** is fixed to known landmarks; but one could conceive using ML to track a custom image (though Snap now covers marker tracking). - **Language/Voice processing:** With voice ML (in Snap's Voice ML category), you could do things like speech recognition or classification of audio input, though that's a specialized area.

To use SnapML: you import the .onnx file (Lens Studio will automatically optimize it a bit). Then add an **ML Component** to an object, assign the model, and choose the input (camera image or other) and output target. For example, a model that outputs a segmentation mask will be set to output as a **Segmentation Texture** (which you can then use like built-in ones) [171]. A model that outputs numbers can be handled via script (the ML Component can provide output to a script as an array or value). Note that heavy models can impact performance, and they consume extra memory (hence Snap's separate ML size budget of 10MB).

Snap also provides some ready-to-use ML models in the Asset Library (e.g. hand gesture recognition model, or a pet face detector, etc.). Check **Lens Studio > Asset Library > ML Models**. This can save you training your own if a common one exists.

Voice ML: This is a subset of ML specifically for audio. Snapchat can do basic speech recognition to trigger lenses based on keywords (e.g. user says "Wow" and lens reacts). As a creator, you might not need to train models for this – Snap has built-in voice commands support for some phrases. But if needed, you could integrate with an external voice API or custom model offline. Voice ML category in LS also includes fun things like transforming the user's voice (like voice changer). The **Audio Effect** component includes options like pitch shifting etc. [107]. If your lens includes a microphone-driven feature, you must use Voice ML component which will handle prompting user for mic permission (and note: if user doesn't grant, your lens should still function gracefully without it).

Connected Lenses (Multi-User AR): This is a fairly advanced Snap feature where two or more users can join the same AR session and see/interact with shared AR objects. For example, two friends could play a tug-of-war game in AR or build a painting together. Implementing this requires using Snap's **Connected Lenses API**, which involves the Lens Cloud (networking) to sync data between devices. Lens Studio has templates for Connected Lenses which handle the low-level networking – you typically just have to decide what state to sync (e.g. positions of shared objects, game scores). Given the complexity, if you want a multi-user lens, definitely start from the Connected Lens template. Just be aware that both users must have the lens open simultaneously and both online; Snap's backend brokers the connection. This is more relevant for games or collaborative experiences. If your concept benefits from multi-user (e.g. a virtual AR ball that two people toss to each other), consider exploring this.

Lens Cloud – Storage and Services: Lens Studio introduced **Lens Cloud** services including: - **Persistent Storage:** You can save data between sessions of a lens or for a specific user (like high scores, or customizations). Useful if you want the lens to "remember" something. It's basically a tiny cloud database keyed per user/lens. Snap's API provides functions to write/read keys [173]. - **Location AR (Custom Landmarkers & GeoLocation):** If you want a lens to trigger at certain locations or use the user's location, Snap has a **Location API**. For example, you could have different lens content if user is at Eiffel Tower versus at home. There's an API to get device lat/long (with user permission). - **Scan API (Visual Search):** Snapchat's Scan can recognize known categories (e.g. scanning a car shows info). As a lens creator, you can tap into Snap's Scan results to augment. For instance, if Snap Scan identifies a dog breed through the camera, your

lens could display additional AR content specifically for that breed. This is more of an advanced integration and might not be fully open to creators yet (some might require partnership).

Remote APIs (HTTP calls): The user's question specifically mentions integrating external data sources and API calls. Yes, **Lens Studio can call external web APIs** (with some restrictions) using the **Remote Service Module** and **Networking API** [174] [175]. For example, you could fetch the current weather from an API and have your lens show the weather info in AR (like an AR weather dashboard). In fact, Snap provides built-in modules for certain popular APIs as examples – e.g. **AccuWeather API** (for weather data) [176], **Stock prices (Alpaca API)** [177], **Snapchat Places API** (for local place info) [177], and even **OpenAI's ChatGPT API** [178]. They have created modules so that you don't have to write raw HTTP code; you import the service module from Asset Library and then just call a function to get the data [174] [175]. For example, the **ChatGPT example** module likely allows the lens to send a prompt to OpenAI and get a response (with certain safety considerations).

To use these, you must import the specific Remote API module from the Asset Library, then in script use the provided interface (the docs often have usage samples). The remote module handles the network request under the hood (since lens JS cannot directly do arbitrary web requests due to security; Snap's modules proxy through Snap's servers). There are best practices: keep such calls minimal and asynchronous, handle no-network scenarios, and note latency (a network call might take a second or two – you don't want your lens freezing in the meantime, so use callbacks as shown in docs [179]). Also, abide by rate limits of the APIs and any API keys needed (some modules like weather might require you to provide an API key from the provider; Snap's examples might show how to include it safely via their Remote Service config).

Potential use-cases: - **Live Data Lenses:** e.g. lens that shows live cryptocurrency prices – it can call a crypto API every few seconds and update a 3D chart. - **Dynamic Content:** e.g. lens that shows a random joke fetched from an API each time. - **User-generated content:** In theory, could fetch images or text that users submitted elsewhere – but careful with moderation. Snap's policies likely forbid un-moderated content display, so ensure anything you fetch is either benign or moderated.

Snap encourages using these capabilities to make lenses more dynamic and context-aware, but they also note it can increase latency and complexity [180], so plan accordingly. They even have a form to suggest new API integrations [181], showing they're open to adding more built-in modules if creators want them.

When using external data, always have a fallback so the lens still functions if the API call fails (for example, if no network, your weather lens could just display a message "No internet – cannot load weather" gracefully, rather than just doing nothing).

Putting Advanced Features together: An ambitious lens might use multiple advanced features. For example, imagine an educational lens that: - Uses **object recognition ML** to identify a plant in the view. - Uses **Remote API** to get information about that plant species from Wikipedia. - Then uses **Text-to-Speech** (Voice ML or an API) to speak out facts to the user, and - Stores which plants the user has identified in **Persistent Storage** to track progress. This would combine SnapML, remote calls, audio, and storage – entirely possible in Lens Studio but requires careful design and testing.

Another example: a multi-user game lens that: - Uses **Connected Lenses** for two players, - Uses **Physics** for game mechanics, - Fetches a daily puzzle using **Remote API**, - And uses **Leaderboards** (Lens Cloud feature) to rank players globally. This is like a full mini-app inside Snapchat.

One more advanced area: **Plugins** – Snap allows developing **Lens Studio Editor plugins** (in Python) to automate tasks or add new tools to the editor [182] [183]. This is more for improving your workflow (like a plugin to batch import something or procedural generation of assets). It doesn't directly affect the lens runtime, but as a power user it can be valuable if you find repetitive tasks in the editor.

Staying Up-to-date: Advanced features evolve. As of 2026, AR is rapidly advancing with things like generative AI integration (Snap's GenAI Suite hints at possibly generating assets or icons on the fly [184] [185]). Keep an eye on Lens Studio release notes and Snap's developer blog for new advanced features. For instance, Snap might introduce new ML model types (maybe 3D mesh reconstruction or advanced face mesh deformation using neural nets), or new API partnerships (maybe news APIs, sports scores, etc.). Being early to use a new feature can make your lens novel.

Finally, **documentation & support:** For these advanced topics, read the official docs (we've cited many sections) and check tutorials. Snap often releases official tutorial videos for major features (like a "How to use VoiceML" video, or a "Build a Connected Lens game" code lab). Also, engaging with the community – if you have a specific crazy idea, someone on the Lens Studio forums/Discord might have insight if it's feasible and how to approach it.

This concludes our deep dive crash course. We covered everything from basic setup to publishing, and touched on cutting-edge features. By now, you should have a strong understanding of Lens Studio's major workflows and capabilities. The platform is very powerful and continually growing – so keep experimenting, learning from others' lenses, and most importantly, **have fun creating**. Lens Studio combines art, code, and interactivity in a unique way, and with the knowledge from this guide, you're well-equipped to build lenses ranging from simple selfie filters to interactive multi-user AR games. Happy lens building!

Sources: The information above references Snap's official Lens Studio documentation and guides for accuracy and up-to-date practices [16] [186] [101] [187] [188] [174], among others. Always refer to Snap's latest docs for detailed APIs and any updated limits or features as Lens Studio evolves.

---

[1] [64] [65] Lens Studio | Snap for Developers
https://developers.snap.com/lens-studio/home

[2] [37] [66] [95] Lens Studio v5.0.10
https://ar.snap.com/download/v5-0-10

[3] [4] [15] [28] [29] [129] Lens Studio Home Page | Snap for Developers
https://developers.snap.com/lens-studio/lens-studio-workflow/lens-studio-interface/home

[5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [30] [44] [45] Panels | Snap for Developers
https://developers.snap.com/lens-studio/lens-studio-workflow/lens-studio-interface/panels

[16] [24] [25] Combining World and Face | Snap for Developers
https://developers.snap.com/lens-studio/lens-studio-workflow/scene-set-up/combining-face-with-world

[17] [80] [81] [82] [83] [84] [85] World AR Overview | Snap for Developers
https://developers.snap.com/lens-studio/4.55.1/references/guides/lens-features/tracking/world/overview

18 19 20 21 22 23 171 172 186 Fullscreen Segmentation | Snap for Developers
https://developers.snap.com/lens-studio/features/ar-tracking/body/segmentation/fullscreen-segmentation

26 27 78 79 Marker Tracking | Snap for Developers
https://developers.snap.com/lens-studio/features/ar-tracking/world/marker-tracking

31 42 43 Overview | Snap for Developers
https://developers.snap.com/lens-studio/features/ui/overview

32 33 34 35 Overview | Snap for Developers
https://developers.snap.com/lens-studio/lens-studio-workflow/scene-set-up/camera

36 Light and Shadow - Snap for Developers
https://developers.snap.com/lens-studio/features/graphics/light-and-shadow

38 70 86 87 Face Effects Overview | Snap for Developers
https://developers.snap.com/lens-studio/features/ar-tracking/face/face-effects-overview

39 40 Prefabs | Snap for Developers
https://developers.snap.com/lens-studio/lens-studio-workflow/prefabs

41 130 131 134 135 137 3D Text | Snap for Developers
https://developers.snap.com/lens-studio/examples/lens-examples/3d-text

46 47 48 49 53 57 59 60 63 Scripting Example | Snap for Developers
https://developers.snap.com/lens-studio/features/scripting/scripting-example

50 58 Scripting Introduction | Snap for Developers
https://developers.snap.com/lens-studio/features/scripting/scripting-introduction

51 52 54 55 56 61 Script Events | Snap for Developers
https://developers.snap.com/lens-studio/features/scripting/script-events

62 99 100 101 102 103 104 105 106 115 145 Audio Component | Snap for Developers
https://developers.snap.com/lens-studio/features/audio/playing-audio

67 68 88 89 90 91 92 93 94 97 98 184 Welcome to Material Editor | Snap for Developers
https://developers.snap.com/lens-studio/features/graphics/materials/material-editor/welcome-to-material-editor

69 SceneEvent | Lens Scripting API - Snap for Developers
https://developers.snap.com/lens-studio/api/lens-scripting/classes/Built-In.SceneEvent.html

71 Eye Color | Snap for Developers
https://developers.snap.com/lens-studio/features/ar-tracking/face/eye-color

72 73 74 75 76 77 3D Body and Hand Tracking | Snap for Developers
https://developers.snap.com/lens-studio/features/ar-tracking/body/object-tracking-3d

96 Welcome to Material Editor - YouTube
https://www.youtube.com/watch?v=7CIJ2oG7Rdk

107 Audio Effect | Snap for Developers
https://developers.snap.com/lens-studio/features/audio/audio-effect

108 109 110 111 112 113 Audio Tracks | Snap for Developers
https://developers.snap.com/lens-studio/features/audio/audio-track-assets

114 Audio Analyzer - Snap for Developers
https://developers.snap.com/lens-studio/features/audio/audio-templates/audio-analyzer

116 117 118 119 120 121 122 123 Touch and Interactions | Snap for Developers
https://developers.snap.com/lens-studio/features/scripting/touch-input

124 125 126 127 128 173 Physics Overview | Snap for Developers
https://developers.snap.com/lens-studio/features/physics/physics-overview

132 133 169 170 182 183 Building Games on Snapchat | Snap for Developers
https://developers.snap.com/lens-studio/features/games/games-overview

136 Segmentation - Snap for Developers
https://developers.snap.com/lens-studio/4.55.1/references/templates/face/segmentation

138 139 140 141 161 165 166 167 168 185 Configuring Project Settings | Snap for Developers
https://developers.snap.com/lens-studio/publishing/configuring/configuring-project-info

142 143 144 151 155 Compression | Snap for Developers
https://developers.snap.com/lens-studio/publishing/optimization/overview

146 147 150 154 157 158 159 160 188 Pairing to Snapchat | Snap for Developers
https://developers.snap.com/lens-studio/lens-studio-workflow/pairing-to-snapchat

148 149 152 153 156 187 Performance and Optimization for Lenses | Snap for Developers
https://developers.snap.com/lens-studio/publishing/optimization/performance-optimization-guide

162 163 164 Creating an Icon | Snap for Developers
https://developers.snap.com/lens-studio/publishing/configuring/creating-an-icon

174 175 176 177 178 179 180 181 Remote Service Module | Snap for Developers
https://developers.snap.com/lens-studio/features/remote-apis/remote-service-module