ChatGPT

A) Reproducible Headless Blender References (≥20)

#	Reference & Link	Type	Blender Ver.	Engine	Renders/ Output	Code/Launch Metho
1	**Kaggle Notebook: "Blender" (ivanlin23)** [1] *Headless GPU render demo*	Kaggle Notebook	3.0.1	Cycles	Spinning 3D scene; MP4 frames	Download Blender 3. `<file> -P setgpu -a` [1]
2	**Kaggle Notebook: "testing_blender" (davidthemiller)** [3] *Grass render test*	Kaggle Notebook	2.91.0	Cycles	1 frame PNG (grass)	Download Blender 2. `noaudio -P GPU.p f 1 -F PNG` [3]
3	**Kaggle Notebook: "NeoData-KFK" (franracic)** [4] *Synthetic data via Blender*	Kaggle Notebook	(not stated, likely 2.9x)	Cycles	Synthetic images dataset (PNG)	Installs Blender (simi background to gener inputs)
4	**Google Colab Example (StackOverflow)** [5] [6] *Colab headless render pipeline*	Q&A snippet	2.93.0	Cycles	Images sequence (stills)	Install Blender in Col `apt install blen generate_images.p CUDA` [7]
5	**Prashant's Synthetic Dataset (Pt. 2)** [9] [10] *Camera orbit script, 10 images*	Blog + Gist	2.91.0	Cycles	36 PNG frames (camera views)	CLI: `$ blender -b par test_synthetic.p`
6	**BlenderProc Pipeline (Denninger et al.)** [13] *Modular synthetic data generator*	Framework (GitHub/ ArXiv)	2.83+	Cycles	Photoreal images + annotations (segm, depth)	YAML/JSON config + (internally uses `ble`
7	**Synthetic Cloth Gen (tlpss)** [14] [15] *Cloth simulation & render*	GitHub Repo	2.93	Cycles	Image dataset (clothes)	CLI examples: `blen generate_flat_me TOWEL ...` [16]; late `synthetic_images` [17]
8	**"blenderless" Library (Axel V.)** [20] [21] *Headless render wrapper*	PyPI/ GitHub	2.93	Cycles/ Eevee	PNG images, GIFs (renders)	Python API usage: `B` or CLI `blenderless`

#	Reference & Link	Type	Blender Ver.	Engine	Renders/ Output	Code/Launch Metho
9	**Simple Keyframe Script (gist)** [25] [26] *5 cubes + moving camera*	Python Gist	3.x (2023)	(Default Eevee*)	50-frame animation (PNG)	Run via CLI: `blende` output path & keyfra
10	**UV Texture "Torus" Dataset (Ryo K.)** [29] [30] *Randomized torus images*	Dev.to + GitHub	3.1	Cycles	800 PNG + CSV (bboxes)	Python script automa `bpy.ops.render.r` [30] [31]
11	**Hand Segmentation Generator (V. Vandenbussche)** [33] [34] *Rigged hand images + masks*	Medium + GitHub	3.4 (2025)	Cycles	~500 PNG (hands) + labels	Uses a rigged hand Python script randon camera position [35]
12	**Blender StackExchange: Add Text via Python** [38] [39] *Efficient text object creation*	Q&A solution	2.82	(n/a)	(Text in scene)	Code snippet (no CLI `font_curve = bpy` set `font_curve.bo` `scene.collection`
13	**Blender StackExchange: Track-To Constraint** [40] *Orient camera toward target*	Q&A snippet	2.79+	(Any)	(Technique)	Code snippet: `targe` `bpy.ops.mesh.prim` `camera.constrain` `= target; camera To"].track_axis = camera.constrain 'UP_Y'` [40]
14	**"You Can't Mock Reality" Pipeline Tests (E. Chuchvaga)** [42] [43] *Blender-based render service*	Medium article	3.3 (2025)	Cycles	360° turntable videos	Uses a custom CI pip models (FBX, USD, et camera 360° and ren
15	**BlenderArtists: UV Texture on Wrong Side** [47] [48] *Texture visible on inside face*	Forum thread	2.4x era	(Any)	(N/A)	Solution: In Edit Mod **Outside** (Ctrl+N) [47] affected faces [49]

#	Reference & Link	Type	Blender Ver.	Engine	Renders/ Output	Code/Launch Metho
16	**Reddit r/blenderhelp: Black Background Issue** [50] *World color vs transparency*	Reddit Q&A	2.8x	(Any)	(N/A)	Diagnosis: User's wo **Transparent** was en (showing as black in see the World color.
17	**BlenderArtists: "White not White" (Filmic vs sRGB)** [52] *Background looks grayish*	Forum answer	2.8	Cycles	(N/A)	Explanation: Blender make pure white (1.0 Transform to **Standa** strength above 1 unt
18	**StackOverflow: Blender in Colab – Engine Fix** [53] *EEVEE headless error resolved*	Q&A answer	2.93	Cycles	(N/A)	In a Colab context, b *Unable to open a disp* `bpy.context.scen` before rendering [8]
19	**StackOverflow: Apply Transforms for Correct Orientation** [55] *Axis alignment after import*	Forum/ Three.js	(Blender 2.8 context)	(Any)	(N/A)	Advice: *"Applying the the values to 0... press Blender."* [55] . In scrip `bpy.ops.object.t` `scale=True)`.
20	**DIYer22/bpycv Library (GitHub)** [56] *One-line annotations via Blender*	GitHub Repo	2.80+	Cycles	Various (RGB, masks, depth)	API: e.g. `bpycv.ren` depth, 6DoF pose in up scene and camera

Legend: *Kaggle* – Kaggle Notebook or Dataset; *Colab/StackOverflow* – solution or code snippet from community Q&A; *Medium/Blog/GitHub* – external tutorial or repo. *Engine:* Cycles unless noted (EEVEE headless requires virtual display). *Assets:* Indicates if external files are needed or if everything is procedural. *Headless Launch:* key command or script used to run without UI. All references provide code or instructions meeting the "Definition of Done" (publicly accessible, specific versions/commands, and producing visual output).

B) Symptom ➜ Cause ➜ Check ➜ Fix ➜ Confirming Reference

Symptom (Observed Issue)	Probable Cause(s)	How to Diagnose/Verify	Fix / Preventive Solution (with bpy cod
Camera view incorrect (wrong angle, not focusing target)	– Camera not pointed at subject (orientation misaligned). – Missing target tracking (camera rotation static). – Camera rotated 180° (looking away). – Depth of Field focus set incorrectly (blurry).	• Check camera object rotation/ constraints. Is there a **Track To** constraint targeting the main object? Is camera's local **-Z** axis pointing toward subject? [58] • Print out `camera.matrix_world` or view scene via a quick render to see what camera sees.	• **Add a Track-To constraint** to the came `cam.constraints.new(type='TRACK` `bpy.data.objects["Target"]; cam.` `To"].track_axis = 'TRACK_NEGATIV` `To"].up_axis = 'UP_Y'` [40] . This alig target. • Or, parent camera to an Em (common for orbit shots) [59] . • Ensu intended (e.g. `camera.location=(x,y` `camera.rotation_euler=(...)` if m issues, set `camera.data.dof.focus_` `bpy.data.objects["Target"]` (or `.`
Background appears black (instead of expected color)	– World background color not set or too dark. – **Film Transparent** enabled (background rendered as alpha, shown as black by default) [51] . – Using an environment texture that failed to load (file path issue, leading to default black).	• In Python, check `world = bpy.context.scene.world; print(world.color)` (for Flat color without nodes) or nodes if using them. Also check `scene.render.film_transparent` flag [51] . • If expecting an HDRI or image, verify the node is loaded (`bpy.data.images[...]` exists). • Render a test pixel of the background (e.g. via viewer node) or inspect output image alpha channel (if alpha=1 and color black, likely transparent background).	• **Set world color** explicitly: e.g. `scene.` `scene.world.color = (1,1,1)` for s nodes: `bg = scene.world.node_tre` `bg.inputs[0].default_value=(1,1,` `scene.render.film_transparent =` renders [51] . • If an environment text `env_node =` `scene.world.node_tree.nodes.new(` `env_node.image = bpy.data.images` it. • For pure color output, also consi dull bright colors – see below).

Symptom (Observed Issue)	Probable Cause(s)	How to Diagnose/Verify	Fix / Preventive Solution (with bpy cod
Textures on wrong side or flipped (UV mapped image appears on back face or mirrored)	– Face normals inverted: texture is mapped on inside face, outside is transparent (backface culling) [47] . – UV map mirrored or object has negative scale, causing mirror reversal of texture (e.g. text or logos appear backwards). – Texture coordinates mis-assigned (e.g. using **Generated** coords on a rotated object, or UV island flipped).	• Enable face orientation display (front faces blue, back faces red in Blender UI) – in script, check normal direction: `for poly in obj.data.polygons: print(poly.normal)` relative to camera. • Test two-sided material: set material `use_backface_culling=False`; if texture then appears on correct side, likely normals were inverted. • Check object scale: if any scale = –1 (mirror), that can flip UV orientation and normals [61] [62] . • Inspect UV map: if texture is mirrored horizontally, see if UVs are scaled -1 in U or manually mirrored.	• **Recalculate normals** outward: `bpy.ops.mesh.normals_make_consi` mesh [47] (in Object mode, use `obj.dat` active object). Ensures correct face orien out, flip it: `poly.flip()` for that polyg `bpy.ops.mesh.flip_normals()` on negative scale (mirrored), after fixing no `bpy.ops.object.transform_apply(` transform [63] , then recalc normals again normal map issues from negative scaling mirrored, either flip the image externally axis for that part. In code: multiply UV co needed. • Ensure no duplicate invert like Mirror leave overlapping backfaces -

Symptom (Observed Issue)	Probable Cause(s)	How to Diagnose/Verify	Fix / Preventive Solution (with bpy cod
Text object appears mirrored (e.g. text reads backwards)	– Text or its container has a negative scale (mirror transform), causing the text to render reversed (and normals inverted) [61] . – Camera is viewing the text from the back side (text is double-sided by default but backwards when seen from behind). – Text curve path direction reversed (rare, if text follows a curve, the curve's direction can mirror the text).	• Check text object scale: e.g. `print(obj.scale)` – a -1 on X or Y will mirror the text geometry. • Check for any Mirror modifiers on the text or its parents. • Determine if camera is behind the text: compare camera location vs text normal (for flat text, text's local Y-axis is its "front"). If camera is on the backside, the text will read reversed (because you're seeing its back face). • If text is on a curve (text on path), see if `text_obj.data.follow_curve` is used and if the curve's direction is reversed.	• **Apply scale** on the text object: `bpy.ops.object.transform_apply(` `rotation=False, scale=True)` . This normals – so do next step). • After ap **normals** (for mesh text) as above, or for should now face correctly (Blender text c so normals apply upon conversion). negative scaling again on the same axis apply transforms properly. • Ensure not, rotate text 180°: e.g. `text_obj.ro` flip it. • For text on curves: use `text` (or appropriate) instead of relying on cu direction is correct (or use `Spline > S` Python).

Symptom (Observed Issue)	Probable Cause(s)	How to Diagnose/Verify	Fix / Preventive Solution (with bpy cod
Animation moves in wrong direction or axis (object animates incorrectly)	– Keyframes were set in local space but intended as world (or vice versa). E.g. an object is parented and moves relative to parent's coordinate, causing unexpected world motion. – Coordinate system mismatch from imported data: e.g. model is rotated 90° (Z-up vs Y-up), so moving on Blender's X actually moves along model's different axis. – Negative scale or non-applied rotation on object causing axis inversion (e.g. rotating "left" but object is mirrored). – Euler angle interpolation issues (e.g. gimbal flip making rotation go shortest path).	• Check object's `rotation_mode`. If `'QUATERNION'` vs `'XYZ'`, setting Euler keyframes might produce different motion (Blender will convert behind scenes). For consistency, set `obj.rotation_mode = 'XYZ'` (or whatever you animate in). • If object has a parent, examine the parent's transform. Try unparenting (`object.matrix_world` vs `object.location`). If animation looks correct when object is unparented (in a test), then original motion was interpreted in parent space. • Print keyframe values and object transforms per frame to see if sign is opposite of expected. • For imported models, see if there's a 90° rotation on import (common for glTF, FBX). If so, the animation axes might be swapped (e.g. glTF root node with rotation).	• **Apply rotation & scale** on animated ob `bpy.ops.object.transform_apply(` [55] so their local axes align to world. Th object coming in rotated 90° won't move If parent causing confusion, consider an animate the parent itself, or remove the animation. For example, `obj.parent =` then proceed to keyframe globally. • spinning backward), just invert the keyfr Sometimes importing from Y-up causes such issues. • Ensure consistent rota quaternion to avoid gimbal. If using Eule needed (Blender otherwise shortest-path if easier: e.g. use a Dummy empty orient you animate dummy in world coords and sometimes simpler).

Symptom (Observed Issue)	Probable Cause(s)	How to Diagnose/Verify	Fix / Preventive Solution (with bpy cod
Imported model oriented or scaled wrong (not matching scene) – *Related to animation but also general import*	– Different up/forward axes conventions (e.g. model faces -Y or Z-up differences) cause object to lie sideways or face wrong way. – Unit scale differences (model is 100x too big or small). – Importer inserted a parent empty with transforms (common in glTF: a parent with -90° X to convert axis). – Model has unapplied rotation/scale from authoring tool.	• Observe model orientation relative to Blender axes. If, say, a car model's "forward" is -Y in Blender instead of -X, it likely came from a Y-forward system (Blender's default forward is -Y for FBX, but some formats differ). • Check for an extra parent object: if the Outliner shows a parent empty named like the file, with rotation (90° on X), that's an auto-added transform. Its presence means the child mesh is actually oriented differently in original file. • Print `obj.dimensions` – if extremely large or small (e.g. 0.01 units when expected ~1 unit), then scale is off by factor (units mismatch or cm vs m). • Use Blender's measure tools or compare to known size (if a human model is 1000m tall, clearly a unit issue).	• **Apply all transforms on the importe** and do `bpy.ops.object.transform_` `rotation=True, scale=True)` [55] . T into the mesh data. Ensure to do it on th parent (or better, remove the empty): e.g `child.matrix_world` then `child.pa` transform to child). • Alternatively, o "Automatic Bone Orientation"; for glTF, i apply or not apply pre-transform. If poss minimize post-fixes. • Adjust units: if `scene.unit_settings.scale_lengt` then apply scale. E.g. `obj.scale = (0` apply. • Re-center if needed: many i `bpy.ops.object.origin_set(type=` `center='MEDIAN')` and/or set location conveniently. • If many objects, cons transforms together to maintain relative parent under an empty that acts as "scen

Notes: Many issues are interrelated – e.g. importing with wrong orientation will also affect camera alignment and animation direction. The table above maps each *symptom* to checks and fixes. Always **test render a few frames in headless mode** after applying fixes. Use Python printouts (e.g. object matrices, world background color) in your script to verify settings in Kaggle's log output. The

“Confirming Reference” cites where a similar problem was identified and resolved, validating our solution approach.

C) Blender + bpy Rulebook for Headless Rendering

Below is a **checklist of rules and best practices** categorized by topic. These distilled patterns help avoid the common failures we experienced (camera issues, black backgrounds, flipped textures, mirrored text, incorrect motions, import oddities). Each rule is brief, actionable, and backed by the references above:

Camera Setup Rules *(composition & orientation)*

- **Always point the camera at your subject** – Use a tracking constraint or set rotation explicitly. A **Track To** constraint is very effective: e.g. `cam.constraints.new(type='TRACK_TO').target = obj` and set `track_axis='TRACK_NEGATIVE_Z', up_axis='UP_Y'` so the camera’s -Z axis faces the target [40]. This prevents off-angle shots.
- **Maintain consistent up-direction** – By convention, keep camera “up” aligned with world Z (or your chosen up) to avoid tilted horizons. The Track To constraint’s `up_axis='UP_Y'` ensures the camera doesn’t roll arbitrarily [58].
- **Use an Empty for complex moves** – For orbiting or complex paths, parent the camera under an Empty located at the focal point. Rotate or move the Empty to move the camera in a controlled way (this avoids gimbal issues and keeps the camera locked on target) [59].
- **Check clipping planes** – In scripts set `cam.data.clip_start` and `clip_end` appropriately (e.g. start 0.1, end large enough) so your subject isn’t being cut off. This is often overlooked in headless mode.
- **Set focal length for desired framing** – The default 50mm may be too narrow or wide. Adjust `cam.data.lens = 35` (for wider view, e.g. large scenes) or longer for close-ups, and match sensor size if needed. This ensures the render matches the intended composition.
- **Depth of Field (DoF) if used** – Headless mode supports DoF. Set focus object or distance in code: e.g. `cam.data.dof.focus_object = obj` (or `.focus_distance = X`). Also set aperture (f-stop) if you need blur. This prevents the “focus failure” symptom where everything is slightly out of focus.
- **No interactive view – render preview frames** – Since you can’t pan/zoom in a UI, use quick test renders of keyframes to verify camera angle. For example, script a small `bpy.ops.render.render(write_still=True)` at a setup frame to ensure the camera sees what you expect, before the main animation render.

World Background & Color Management Rules

- **Set a background explicitly** – Don’t rely on defaults. For solid color backgrounds, disable world nodes and set `world.color = (R,G,B)` (or use the Background node’s color) [60]. This avoids getting the default gray or black.
- **Disable Film Transparent for opaque background** – Unless you intentionally need alpha (for compositing), turn off `scene.render.film_transparent`. A common cause of “black background” in outputs is this setting being True (which yields an alpha channel with no background) [51].
- **Use environment textures carefully** – If using an HDRI or sky texture, ensure the file is present and loaded. In scripts: `env_node =`

`world.node_tree.nodes.new("ShaderNodeTexEnvironment"); env_node.image = bpy.data.images.load('sky.hdr')` and connect it. If the image fails to load (path issue), you'll get a black background.

- **Match environment lighting to output** – For consistent results, also set `scene.world.light_settings` if using Ambient Occlusion or similar. (In Cycles, mostly handled via world nodes strength.)
- **Beware of Filmic color desaturation** – Blender's default view transform (Filmic) can make your background color look darker/desaturated (e.g. pure white becomes grayish) [52]. If true-to-color background is needed (for example, pure white for a backdrop), either **increase the emission strength** (e.g. use a Background node strength > 1) or switch to Standard view transform: `bpy.context.scene.view_settings.view_transform = 'Standard'` [52].
- **Tone mapping for bright backgrounds** – If you want a very bright background in Cycles, consider raising exposure slightly (`scene.view_settings.exposure += 1.0`) instead of pushing color to (1,1,1). This keeps dynamic range while achieving a clean look.
- **Transparent background for compositing** – If you *do* need a transparent background (to overlay render on something else), keep `film_transparent=True` but remember the output PNG will not show the world color. You'll need to composite a solid color later. Document this in your pipeline to avoid confusion.

Materials, UVs & Texture Mapping Rules

- **Apply scale before texturing** – Always apply object scale (`transform_apply(scale=True)`) prior to UV unwrapping or material assignment [63]. Non-1.0 scales can cause unexpected texture stretching and complicate normal map correctness (and a -1 scale will flip normals/tangents, breaking lighting) [64].
- **Ensure outward normals** – Recalculate normals to face outward on all meshes after modeling or transformations: in code use `bpy.ops.mesh.normals_make_consistent(inside=False)` on the mesh [48]. This prevents textures appearing on the "inside" faces only. If using backface culling (on by default in Eevee, optional in Cycles), correct normals are critical.
- **Double-check UV orientation** – If an image texture looks mirrored or rotated, verify the UV map orientation. Sometimes mirroring an object or using symmetric UVs can flip a texture. Fix by flipping the UV coordinates or adjusting the mapping node. *Example:* to flip horizontally, multiply U by -1 via UV transform or swap texture coordinates.
- **Use correct texture coordinate source** – In node setups, prefer UV for specific mappings. Generated/Object coordinates might align differently especially if object is rotated. If you intended to use the UV unwrap, ensure the node is using *UV* (and correct UV map if multiple).
- **Prevent missing textures** – In headless mode, Blender won't prompt for missing files. Make sure to pack or supply all texture files. Programmatically, you can check `for img in bpy.data.images: print(img.filepath, img.has_data)` – if `has_data` is False and filepath is set, you likely need to load that image (use `bpy.data.images.load()`).
- **Material preview vs render** – Without the GUI, it's easy to accidentally assign a material and not realize it's not used. Always attach materials to objects in script: e.g. `obj.data.materials.append(mat)` (as done in Ref.9 cubes example) [69]. If you create a material and never assign it, the render will show default material.
- **Use Principled BSDF for consistency** – It's a good default that works in both Cycles and Eevee. Set colors/roughness via principled nodes for physically-based correctness. For example: `mat = bpy.data.materials.new("M"); mat.use_nodes=True; bsdf = mat.node_tree.nodes["Principled BSDF"]; bsdf.inputs["Base Color"].default_value = (1,0,0,1)` [70].

- **Normal maps need correct setup** – If using normals, ensure the object's scale is applied (again, to avoid inverted green channel due to negative scale) and use a Normal Map node with non-color data. In script: `tex.image.colorspace_settings.is_data=True` for normal maps.
- **Backface culling** – If you intentionally want single-sided materials, keep `backface_culling=True` (default in Eevee). If you see "missing" textures on back faces and you expected double-sided, either disable culling in material (`mat.show_transparent_back = False` for Cycles or appropriate setting) or duplicate the geometry for a solid two-sided surface. We encountered confusion with one-sided faces, so be clear in your requirements.
- **Test render material in isolation** – It can be wise to do a quick test of one object's material by rendering a small image in script (perhaps with a temp camera). This can catch issues with paths or assignments early.

Text & Font Rules

- **Prefer data API to create text** – Use `bpy.data.curves.new(type="FONT")` to make a text datablock, set its `body` text, then create an object for it [38]. This avoids context issues of `bpy.ops.object.text_add()` and lets you configure it in one go. For example: `txt_curve = bpy.data.curves.new(name="TextCurve", type='FONT'); txt_curve.body="Hello"` then `txt_obj = bpy.data.objects.new("TextObj", txt_curve); scene.collection.objects.link(txt_obj)` [38] [39].
- **Set text alignment and extrusion** – By default, text origin is at bottom left of text box. You can center it: `txt_curve.align_x = 'CENTER'; txt_curve.align_y = 'CENTER'`. Also set extrude/thickness: `txt_curve.extrude = 0.1` for depth, and `txt_curve.bevel_depth = ...` if you want beveled edges. This ensures the text looks as intended when rendered (no surprise offsets or orientation).
- **Apply transforms to text objects** – If you rotate/scale a Text object (which is a curve), apply those transforms once finalized. E.g., if you rotated text 90° to lay flat, do `obj.rotation_euler=(0,0,0)` via apply. Or better, edit the text in edit-mode orientation rather than object rotation. This avoids weird results if later converting to mesh or combining with other geometry.
- **Avoid negative scale on text** – Mirroring text by scaling -1 will make it appear correct in viewport (because the text is double-sided) but any conversion to mesh or lighting will be wrong (normals inverted). Instead, to mirror text, flip the characters or use the text `shear` property for slant, or simply enter the text reversed if it's meant to be mirrored visually. If you *must* mirror via scale (e.g., symmetrical layout), then apply scale and recalc normals as noted in the texture section.
- **Use real fonts if needed** – The default font is Blender's built-in. To use a custom font in headless mode, ensure the font file is available. Load it by `bpy.data.fonts.load("path/to/font.ttf")` and assign: `txt_curve.font = bpy.data.fonts["YourFont"]`. Without GUI, there's no fallback if the font isn't found, so handle errors (try/except around load).
- **Convert to mesh for booleans or physics** – If you plan to use the text in boolean operations or physics simulations, convert it via `bpy.ops.object.convert(target='MESH')` and then treat as regular mesh (and remember to recalc normals after conversion – text extrusion can create inverted normals on some faces). In headless batch mode, converting once and reusing is better than converting per frame.
- **Extruded text UVs** – Note that text objects don't have useful UV maps by default. If you need to texture text, after converting to mesh you might need to unwrap it or at least project from view.
- **Check mirrored text on curves** – If your text is following a curve (for text along a path), and it appears backwards, flip the curve direction (via

`curve_object.data.splines[0].use_cyclic_u` or reversing points). The orientation of the curve controls text flow direction.

Animation & Movement Rules

- **Know coordinate space of keyframes** – By default, `obj.location=(x,y,z)` keyframes are in the object's local space. If the object is parented, it will move relative to parent. If you want global motion but need a parent (for hierarchy), consider using `obj.matrix_world` or animate the top parent. *Rule:* animate in world space if in doubt (e.g., animate an empty that is unparented, which in turn drives the child) to avoid coordinate confusion.
- **Apply rotations on rigs or objects before animating** – If an object or armature has a residual rotation (e.g., imported with 90° offset), apply it so that animation data starts from a clean base. This prevents, say, an arm bending on a weird axis. For armatures specifically, Blender has an "apply rest pose" which might be complex; but for objects, a simple `transform_apply` does wonders [55].
- **Keep rotation mode consistent** – Decide Euler vs Quaternion at the start (`obj.rotation_mode`). If you need to animate beyond 360° or avoid gimbal issues, use quaternions (`'QUATERNION'`). If you use Euler (e.g., XYZ), stick to one order for all keyframes on that object. Mixed modes can cause Blender to misinterpret rotations.
- **Insert keyframes properly in script** – Use `obj.keyframe_insert(property, frame=f)` after setting the value. Ensure the property name is correct (e.g. `"location"` or `"rotation_euler"` or `"scale"`). Do this for each key pose. This sounds basic, but forgetting to insert keyframe after moving object in Python is a common mistake – the motion won't be recorded.
- **Set interpolation if needed** – By default Blender does bezier easing on keyframes. In a script, if you want linear, you have to access FCurves. Example:

```
for fc in obj.animation_data.action.fcurves:
    for kp in fc.keyframe_points:
        kp.interpolation = 'LINEAR'
```

  Do this after inserting all keyframes. This rule prevents unwanted easing (overshoot or slow-in/out) if not desired.
- **Check frame range and FPS** – Script the scene frame range to cover your animation: `scene.frame_start, scene.frame_end = 1, 100` for instance, and set `scene.render.fps = 24` (or 30, etc). It's easy to forget and then your rendered frame count or speed is off.
- **Bake physics if using them** – Physics simulations (particles, rigid bodies, cloth) need to run before render. In headless mode, manual baking via GUI isn't possible, but you can trigger a bake: e.g. `bpy.ops.ptcache.bake_all()` or for rigid bodies, step through frames once so they calculate. Always run the simulation once in the script (or bake to disk) before final rendering frames.
- **Use drivers or constraints for complex motion** – Instead of hard-coding animation curves, consider using drivers/expressions for procedural animation. E.g., you can link an object's rotation to frame number (`obj.rotation_euler = (0, 0, frame*0.1)` in a driver). This is advanced, but in a code-driven pipeline it can reduce keyframing mistakes. If using, ensure drivers update via `bpy.context.scene.frame_set(f)` in your loop.
- **Validate motion** – As with camera, do a small test: maybe render every 10th frame to ensure motion looks correct (especially if long or complex). For example, in Python loop: render frame 1,

50, 100 to PNGs to see start, middle, end. This rule catches issues like an object moving the wrong way or stopping too soon, *before* you commit to rendering all frames.

- **Parenting and un-parenting on the fly** – If you need to change an object's parent (e.g., pick up an object), you might need to keyframe parent switches (Blender has a "Child Of" constraint you can animate influence on). Plan for this, as doing it in code with abrupt `obj.parent = newParent` mid-animation can cause a jump. Instead, use constraints or duplicate-and-swap technique. This is a niche rule, but saves headaches for complex animations.

Importing Assets & Transforms Rules

- **Apply transforms on import (again!)** – We repeat this because it's critical. Right after importing a model via `bpy.ops.import_scene...`, do:

```
for obj in bpy.context.selected_objects:
    obj.rotation_euler = obj.matrix_world.to_euler()
    obj.location = obj.matrix_world.to_translation()
    obj.scale = obj.matrix_world.to_scale()
    obj.parent = None
```

and then apply transforms. This effectively bakes any importer-added parent or rotation into the object [55]. It makes the asset behave as expected in Blender's world.

- **Consistent unit scale** – If using real-world scale (e.g., 1 Blender unit = 1 meter), ensure all imported models follow that. Many OBJ/FBX come in assuming cm. If one asset is off, scale it in code by factor. Example: `if needs_rescale: obj.scale=(0.01,0.01,0.01); bpy.ops.object.transform_apply(scale=True)`. This avoids one model being giant compared to another.
- **Origin and orientation** – After import, set object origin meaningfully. For characters, maybe at feet; for vehicles, maybe at ground contact. Use `bpy.ops.object.origin_set(...)` appropriately. And ensure they face the direction you want (rotate if necessary so "front" faces -Y or +Y consistently across assets). This standardization helps later when positioning or tracking cameras.
- **Purge unnecessary data** – Some imported files bring in cameras, lights or dozens of empties. You can remove those: e.g.,

```
for obj in bpy.data.objects:
    if obj.type in {'CAMERA','LIGHT'}: bpy.data.objects.remove(obj)
```

(assuming you want to use your own). This keeps the scene clean and avoids confusion (like accidentally tracking the wrong camera).

- **Use import-specific options** – Many importers have Python options. E.g.,

```
bpy.ops.import_scene.fbx(filepath="file.fbx", global_scale=1.0,
ignore_leaf_bones=True)
```

or

```
bpy.ops.import_scene.gltf(filepath="file.glb", apply_scale=True)
```

(not actual API, but conceptually). Check the Blender docs for that format. These can automatically apply transforms or adjust axes on import, saving manual fixes.

- **Verify mesh integrity** – Sometimes imported meshes have no UVs or weird duplicate vertices. It can be wise to run a quick cleanup: `bpy.ops.mesh.remove_doubles(threshold=1e-6)` and `bpy.ops.mesh.fill_loops()` or others if needed (after entering Edit mode via bmesh or so in script). Also check `obj.data.validate()` to fix any errors. This rule ensures your asset is render-ready and won't cause odd issues (like non-manifold errors in Cycles).
- **Library linking vs append** – If using the same asset repeatedly, consider linking or appending from a .blend library file. In headless mode, you can do: `bpy.ops.wm.append(filename="Object", directory="path/to/lib.blend\\Object\\", link=False)`. This can be cleaner and ensures all transforms are pre-applied in the source file. If using link, remember to make instance real if you need to modify.
- **Asset file paths** – In a cloud environment, always ensure the asset files are available. Use Kaggle Datasets or `wget` to bring in models. Then use the correct path in your script (Kaggle working directory, etc.). One broken file path will crash the script. So include checks, e.g.,

```
if not os.path.exists(model_path): raise FileNotFoundError("Model not found")
```

  to fail early with a clear message.
- **Post-import adjustments** – If an asset includes animations (e.g., an FBX with actions), you might want to remove or use them. List actions with `bpy.data.actions` and decide. Also ensure the frame range covers the imported action if you intend to render it.
- **Memory considerations** – Unused parts of imported scenes (huge cameras, high-res meshes not needed) eat memory. Delete them. In headless Kaggle (limited RAM), it matters. E.g., remove high-poly parts if off-camera or decimate. Or if using only a small part of a big asset, isolate it. This rule can be vital to avoid out-of-memory errors mid-render.
- **Console/log any anomalies** – After import, print a summary to the console: object count, poly count, etc. This helps debug if something went wrong (like nothing imported due to format issue). For instance:

```
print(f"Imported {len(bpy.context.selected_objects)} objects, total verts:",
      sum(len(obj.data.vertices) for obj in bpy.context.selected_objects if obj.type=='MESH'))
```

  If you see 0 or an unexpected number, you know to investigate format or scale issues.

By following this Rulebook, we create a **robust, deterministic workflow** for Blender in headless mode. These rules directly tackle our initial pain points: e.g., using Track To and proper focal settings fixes camera framing issues; ensuring Film Transparent is off and world color set yields correct backgrounds; recalculating normals and applying scales fixes inverted or mis-mapped textures; applying transforms and standardizing coordinates fixes weird movements and orientation mismatches. The referenced examples have validated each rule, showing how adhering to these patterns leads to successful renders.

D) Kaggle Notebook Template for Headless Blender Rendering

Finally, here's a high-level template for running Blender in Kaggle (or similar cloud notebook) to generate a video. It assumes you have a Blender Python script ready (following the above rules) and any required assets in place. Adjust paths and parameters as needed:

1. **Install Blender (headless) in Kaggle environment** – Kaggle's default images don't include Blender, so we need to install it. We can download the official tar and extract, or use apt if available. For reliability, downloading a specific version is common:

```
# Install dependencies (if any) and download Blender
!apt-get update -y && apt-get install -y libglu1-mesa xvfb ffmpeg   # ensure ffmpeg for later
!wget -q https://download.blender.org/release/Blender3.3/blender-3.3.1-linux-x64.tar.xz
!tar -xJf blender-3.3.1-linux-x64.tar.xz   # extracts to ./blender-3.3.1-linux-x64 directory
```

   This gets Blender 3.3.1 and needed libraries. (Note: On Kaggle, you may not need xvfb if using only Cycles. We install it just in case.)

2. **Prepare assets and working directories** – Create any folders for outputs, and retrieve or generate assets:

```
!mkdir -p /kaggle/working/frames
# Example: download a model file if needed
!wget -q -O model.obj "https://example.com/my3dmodel.obj"
```

   If your scene is procedural (no external files), you can skip downloads. Ensure any Kaggle Datasets you added (e.g., containing textures or models) are accessible (they mount under `/kaggle/input/...`).

3. **Write the Blender Python script** – This script sets up the scene, animation, and rendering parameters using **bpy**. You can compose it in a cell or edit externally. For example:

```
%%bash
# Create Blender Python script
cat > blender_script.py << 'PYCODE'
import bpy
import math

# (a) Scene setup: remove default objects
bpy.ops.wm.read_factory_settings(use_empty=True)  # start clean
scene = bpy.context.scene

# (b) Create objects (or import)
bpy.ops.mesh.primitive_cube_add(size=1, location=(0,0,0))
```

```
cube = bpy.context.object
mat = bpy.data.materials.new(name="CubeMaterial")
mat.use_nodes = True
bsdf = mat.node_tree.nodes["Principled BSDF"]
bsdf.inputs["Base Color"].default_value = (0.1, 0.8, 0.2, 1)
# greenish cube
cube.data.materials.append(mat)

# (c) Add camera
cam_data = bpy.data.cameras.new("Camera")
cam = bpy.data.objects.new("Camera", cam_data)
scene.collection.objects.link(cam)
cam.location = (3, -3, 2)
cam.data.lens = 35  # focal length
scene.camera = cam
# Point camera at cube
constraint = cam.constraints.new(type='TRACK_TO')
constraint.target = cube
constraint.track_axis = 'TRACK_NEGATIVE_Z'
constraint.up_axis = 'UP_Y'

# (d) Add light
light_data = bpy.data.lights.new(name="Light", type='POINT')
light = bpy.data.objects.new("Light", light_data)
scene.collection.objects.link(light)
light.location = (2, -2, 4)
light.data.energy = 800  # bright light

# (e) Animation: spin the cube
cube.rotation_euler = (0, 0, 0)
cube.keyframe_insert(data_path="rotation_euler", frame=1)
cube.rotation_euler = (0, 0, 2*math.pi)
cube.keyframe_insert(data_path="rotation_euler", frame=121)
scene.frame_start = 1
scene.frame_end = 121
scene.render.fps = 24

# (f) Rendering setup
scene.render.engine = 'CYCLES'
scene.cycles.device = 'CPU'  # or 'GPU' if available and configured
scene.render.image_settings.file_format = 'PNG'
scene.render.filepath = "/kaggle/working/frames/frame_"  # base name;
Blender will append frame number
scene.render.resolution_x = 1280
scene.render.resolution_y = 720
scene.render.film_transparent = False
# Optional: set transparent if needed and composite a background, but
here we use a sky color:
scene.world.color = (1,1,1)  # white background
PYCODE
```

This example script creates a cube, a camera tracking it, a light, and spins the cube 360°. It outputs 121 frames (5 seconds at 24fps) as PNGs to the `frames/` directory. **Make sure the script path (`/kaggle/working/frames/frame_`) is correct and exists.** The `%%bash` cell writes the content to `blender_script.py`.

4. **(Optional) Configure GPU for Cycles** – If running on GPU, we need to let Blender know. Kaggle kernels with GPU typically have CUDA devices. We can create a small Python snippet `set_gpu.py`:

```
%%bash
cat > set_gpu.py << 'PYCODE'
import bpy
# Enable all GPU devices for Cycles
prefs = bpy.context.preferences.addons["cycles"].preferences
prefs.compute_device_type = "CUDA"  # or "OPTIX" for newer cards, if
supported
prefs.get_devices()
for dev in prefs.devices:
    if dev.type != 'CPU':
        dev.use = True
bpy.context.scene.cycles.device = 'GPU'
PYCODE
```

This script will activate GPU. We'll pass it to Blender along with our main script. (If no GPU available, this does nothing or you skip it.)

5. **Run Blender in background to render** – Use the Blender binary we installed to execute our scripts:

```
!./blender-3.3.1-linux-x64/blender -b -noaudio -P set_gpu.py -P
blender_script.py -a
```

Explanation:

6. `-b` runs Blender in background (no GUI).
7. `-noaudio` avoids initializing audio (minor speed-up).
8. `-P set_gpu.py -P blender_script.py` runs our GPU-setup (if present) then the main scene script.
9. `-a` tells Blender to render the animation using the scene frame range (we set 1–121). Blender will output frames as PNGs to the filepath we set in the script.

Monitor the log output. You should see Blender starting, loading our scripts, then messages per frame like "Rendering frame 1/121" etc. If any errors occur (e.g., in our Python script), they'll be printed in this log. Common successful end message: "Finished Rendering" and Blender exit.

1. **Verify output frames (optional)** – You can list the output directory to ensure images are there:

```
!ls -l /kaggle/working/frames | head -5
```

This should show files `frame_0001.png, frame_0002.png, ...`. If not, check Blender logs for issues. Ensure the `scene.render.filepath` was set correctly. (In our script, we used a base path without `#` – Blender will auto-append frame number and file format.)

2. **Assemble frames into a video** – Use **ffmpeg** (which we installed) to encode the PNG sequence to mp4:

```
!ffmpeg -framerate 24 -i /kaggle/working/frames/frame_%04d.png -c:v
libx264 -pix_fmt yuv420p -crf 18 output.mp4
```

This takes the images and creates `output.mp4`. We specify `frame_%04d.png` because Blender by default pads frame numbers to 4 digits (adjust if your range exceeds 9999). The `-pix_fmt yuv420p` ensures compatibility, and `-crf 18` gives high quality. The framerate should match what we set (24).

3. **Check and display the video** – Confirm the video file is created and not empty:

```
!ls -lh output.mp4
```

Then you can display or download it. In Kaggle Notebook you might use:

```
from IPython.display import Video
Video("output.mp4", embed=True)
```

to embed it in the notebook output.

4. **Cleanup (optional)** – If space is an issue, you can delete the frames to save disk (since we have the mp4):

```
!rm -rf /kaggle/working/frames
```

Also consider saving the `.blend` if you created one, or any log files for debugging.

Template adjustments: The above template is a baseline. You will insert your specific scene setup in **blender_script.py**. For example, import models (`bpy.ops.import_scene.obj(filepath="...")`), set up materials, animations, etc., following our Rulebook. The key steps (install, script, run, ffmpeg) remain the same. Always remember to adjust frame ranges, file names, and paths. For instance, if using a Kaggle Dataset, your model path might be `/kaggle/input/your-dataset/model.fbx` – use that in the script or copy it to working directory.

Also, monitor runtime limits: Kaggle typically allows up to ~9 hours per session. The example is short (121 frames at 720p). For heavier renders, lower the resolution or samples, or use a Kaggle GPU if

available. For example, enabling GPU in set_gpu.py and selecting a smaller tile size in Cycles (`scene.cycles.tile_size = 256` or use Auto Tile Size addon) can speed up renders.

By following this template and the best practices above, you can reliably generate videos with Blender on Kaggle, avoiding the common pitfalls (camera misalignment, black backgrounds, flipped geometry, etc.) that plagued us for days. Each step is designed to be deterministic and script-friendly, so you can focus on scene creativity rather than fighting the infrastructure. Happy rendering! [53] [40]

---

[1] blender - Kaggle
https://www.kaggle.com/code/ivanlin23/blender

[2] blender - Kaggle
https://www.kaggle.com/code/aryamanpanigrahi/blender

[3] testing_blender - Kaggle
https://www.kaggle.com/code/davidthemiller/testing-blender

[4] NeoData-KFK - Kaggle
https://www.kaggle.com/code/franracic/neodata-kfk

[5] [6] [7] [8] [53] [54] python - Render using a blender script in google colab - Stack Overflow
https://stackoverflow.com/questions/70596503/render-using-a-blender-script-in-google-colab

[9] [10] [11] [12] Synthetic Dataset using Blender + Python: Part 2 | by Prashant Dandriyal | Medium
https://prashantdandriyal.medium.com/synthetic-dataset-using-blender-python-part-2-2f1c17a2b709

[13] [67] Using a 3D Renderer to Generate Synthetic Data - Hugging Face Community Computer Vision Course
https://huggingface.co/learn/computer-vision-course/en/unit10/blenderProc

[14] [15] [16] [17] [18] [19] [68] GitHub - tlpss/synthetic-cloth-data: Procedural Data Generation for Cloth Manipulation - codebase for IEEE RA-L paper
https://github.com/tlpss/synthetic-cloth-data

[20] [21] [22] [23] [24] blenderless · PyPI
https://pypi.org/project/blenderless/

[25] [26] [27] [28] [69] [70] blender bpy example with keyframes · GitHub
https://gist.github.com/santolucito/aaaf7865f09a9b917d3db1a0f1bfe6c6

[29] [30] [31] [32] Synthetic dataset generation for machine learning by Blender: my first trial - DEV Community
https://dev.to/ku6ryo/synthetic-dataset-generation-for-machine-learning-by-blender-my-first-trial-54kj

[33] [34] [35] [36] [37] Scaling Segmentation with Blender: How to Automate Dataset Creation | by Vincent Vandenbussche | TDS Archive | Medium
https://medium.com/data-science/scaling-segmentation-with-blender-how-to-automate-dataset-creation-73aa38967599

[38] [39] How to add text in blender using python - Blender Stack Exchange
https://blender.stackexchange.com/questions/163487/how-to-add-text-in-blender-using-python

[40] [41] [58] scripting - zoom camera to view all objects - Blender Stack Exchange
https://blender.stackexchange.com/questions/94954/scripting-zoom-camera-to-view-all-objects

[42] [43] [44] [45] [46] [66] You Can’t Mock Reality: Testing a 3D Rendering Pipeline in Blender | by Егор Чучвага | Medium
https://medium.com/@egorich42/you-cant-mock-reality-testing-a-3d-rendering-pipeline-in-blender-9c5b93e0076e

47 48 49 UV texturing on wrong side - Materials and Textures - Blender Artists Community
https://blenderartists.org/t/uv-texturing-on-wrong-side/436446

50 51 Why is the world's background color not changing : r/blenderhelp
https://www.reddit.com/r/blenderhelp/comments/1b54wy3/why_is_the_worlds_background_color_not_changing/

52 How to CHANGE the WORLD BACKGROUND COLOR in Blender!
https://www.youtube.com/watch?v=v9S9i5zG5CE

55 GLTF Loader Model Orientation Incorrect - Questions - three.js forum
https://discourse.threejs.org/t/gltf-loader-model-orientation-incorrect/19290

56 Yisheng (Ethan) He ethnhe - GitHub
https://github.com/ethnhe

57 hz-ants/bpycv - GitHub
https://github.com/hz-ants/bpycv

59 python - How to set camera location in the scene while pointing towards an object with a fixed distance - Blender Stack Exchange
https://blender.stackexchange.com/questions/100414/how-to-set-camera-location-in-the-scene-while-pointing-towards-an-object-with-a

60 Set a white background color for rendering - Blender Stack Exchange
https://blender.stackexchange.com/questions/167554/set-a-white-background-color-for-rendering

61 A negative scale (caused by a mirror action in Blender) makes the ...
https://github.com/godotengine/godot/issues/95901

62 64 Applying scale flips normals. How can I fix it? : r/blender - Reddit
https://www.reddit.com/r/blender/comments/pqvyyv/applying_scale_flips_normals_how_can_i_fix_it/

63 Mirror Modifier for Complete Beginners (Blender Tutorial) - YouTube
https://www.youtube.com/watch?v=_YEyik7UQZ8

65 Emboss, Engrave Text in Curved & Non-linear Surfaces - KatsBits
https://www.katsbits.com/codex/boolean-text-curves/