

WebGL and GPU Rendering

I have 2 questions:

1. when I write getcontext as webgl, then the entire bitmap update on each render is taken care by GPU?
2. when i use transform translate or opacity, The element is shifted as new layer and doesnt trigger repaint. But the animation happens. That is taken care by GPU by re-creating the layer on each frame?

Is my observation correct?

Short answer: **you're close, but there are a couple of important corrections.**

Let's go point by point and map this to how browsers actually work.

1. `getContext("webgl")` — is the entire bitmap update handled by the GPU?

Not automatically, and not in the way a normal bitmap is.

When you use **WebGL**:

- You are **not dealing with a browser-managed bitmap** like `<canvas>` 2D.
- You are issuing **GPU commands** (draw calls, buffer updates, texture uploads).
- The GPU renders directly into a **framebuffer** (usually GPU memory).

Key clarifications

- The browser **does not repaint a bitmap for you**.
- **Only what you explicitly change** (buffers, uniforms, textures) is updated.
- If you upload new vertex data or textures each frame, **that upload happens each frame**.
- The GPU does the rasterization and shading, but **you control what changes**.

Example:

```
js
gl.bufferSubData(...) // uploads new data to GPU
gl.drawArrays(...)    // GPU renders using current state
```

So:

- ✗ “*Entire bitmap update is taken care of by GPU automatically*” → **Incorrect**
- ✓ “*Rendering is GPU-driven and incremental based on my commands*” → **Correct**

2. transform: translate() or opacity — GPU recreates the layer every frame?

No — the layer is usually *not recreated*. It's reused and composited.

This is the most common misconception.

What actually happens

When you animate:

```
css
transform: translateX(...)  
opacity: ...
```

The browser pipeline looks like this:

```
sql
Layout      → X skipped
Paint       → X skipped
Composite   → ✓ runs every frame
```

Why no repaint?

- The element is **already painted once** into a bitmap (often called a *layer*).
- That bitmap is **uploaded once as a GPU texture**.
- On each frame:
 - The **GPU repositions or blends the same texture**
 - No pixels are repainted
 - No layout is recalculated

Important correction

- X “GPU recreates the layer each frame” → **Incorrect**
- ✓ “GPU reuses the same texture and applies a new transform or opacity” → **Correct**

Think of it like this:

The pixels stay the same — only the **matrix and alpha values change**

Putting it all together

WebGL

Aspect	Who does it
Drawing pixels	GPU

Aspect	Who does it
Deciding what changes	You
Automatic bitmap repaint	✗ No
Full control	✓ Yes

CSS transform / opacity

Aspect	Who does it
Pixel generation	CPU (once)
Animation	GPU
Repaint per frame	✗ No
Layer recreation	✗ No
Texture reuse	✓ Yes

Mental model that works well

- **WebGL** → You are the rendering engine
- **CSS transforms** → Browser paints once, GPU moves the result
- **GPU animations** → Change math, not pixels

If you want, I can:

- walk through **Chrome DevTools** → Layers / Rendering tabs
- explain **when a new layer is actually created**
- compare **Canvas 2D vs WebGL vs CSS animations**

Just tell me.

GPU reuses the same texture and applies a new transform or opacity" → Correct

but even though GPU will use same texture, after applying transform or opacity, it has to generate a new texture right?

Great question — this is the exact subtle point where a lot of confusion happens 

Short answer: **No, the GPU usually does not generate a new texture.** It generates **new fragments (pixels) in the framebuffer.**

Let's break that apart carefully.

Texture \neq Framebuffer (this is the key distinction)

Texture

- A texture is **source data**
- Stored in GPU memory
- Created once (e.g. from a painted DOM layer or image)
- **Not modified** during transform / opacity animations

Framebuffer

- The **final render target** (what you see on screen)
- Rewritten **every frame** (60fps, 120fps, etc.)
- Built by sampling textures + applying math

What actually happens during a transform/opacity animation

Assume a layer already exists as a GPU texture.

Each frame:

1. GPU takes the **same texture**
2. Applies:
 - a **transform matrix** (translate / rotate / scale)
 - an **alpha value** (opacity)
3. Runs the fragment shader
4. Writes the result into the **framebuffer**

Crucially:

- **✗** No new texture is created
- **✗** Original texture is not rewritten
- **✓** Only the framebuffer pixels change

Why a new texture would be expensive

Creating a new texture would require:

- Allocating GPU memory
- Uploading pixel data
- Syncing CPU \leftrightarrow GPU

That would defeat the entire purpose of compositor-only animations. Browsers explicitly avoid this.

Simple GPU analogy

Imagine:

- **Texture** = a sticker
- **Framebuffer** = a whiteboard

Each frame:

- You move the sticker or change its transparency
- You redraw the **whiteboard**, not the sticker
- The sticker itself never changes

What *does* get regenerated every frame?

Item	Regenerated?
Framebuffer	✓ Yes
Vertex data	✗ Usually reused
Texture	✗ No
Transform matrix	✓ Yes (tiny data)
Alpha value	✓ Yes (tiny data)

When *would* a new texture be created?

A new texture (or repaint → texture upload) happens when:

- Content changes (**text, background, border**, etc.)
- Layout-affecting properties change
- Non-compositor properties animate (e.g. **width, left, box-shadow**)
- Filters that require offscreen rendering (**filter: blur()** often)

In those cases:

```
cpp
Paint → new bitmap → new texture upload
```

Corrected statement you can safely use

"The GPU reuses the same texture; only the framebuffer is re-rendered each frame with updated transform or opacity values."

That's the precise, correct mental model.

If you want, I can also explain:

- why `will-change` helps *sometimes*
- how `filter` differs from `opacity`
- how WebGL's pipeline maps almost 1:1 to browser compositing

Just say the word.

still i dont understand how a image can change in framebuffer using old image. opacity is fine but how transform translate will work? I have a image with a glass on center and now i move the glass towards the right corner. so you are saying the image in frame buffer is same but the entire image is moved and shown as if that one glass moved(a different layer)?

This is a *very* good question — you're not missing something trivial; you're bumping into how GPUs **actually draw images** versus how we intuitively imagine them.

The confusion comes from thinking:

"The image itself must move inside the framebuffer"

That is **not** what happens.

Let's rebuild the picture from first principles.

The crucial idea (one sentence)

The GPU never "moves pixels."

It redraws the framebuffer every frame by sampling the same texture at different screen positions.

Nothing in the framebuffer is reused from the previous frame.

Step 1: Forget the previous frame exists

A framebuffer is not like a Photoshop canvas that gets edited.

Each frame:

- The framebuffer is **cleared**
- The GPU **draws everything again**

So there is no "old image in the framebuffer" being moved.

Step 2: How an image is actually drawn (even without animation)

When the browser draws your image layer, it does this:

1. Creates a **rectangle (two triangles)**
2. Attaches the image as a **texture**
3. For each pixel covered by the rectangle:
 - Sample the texture
 - Write the color into the framebuffer

So the image is *never copied pixel-for-pixel*.

It is **reconstructed every frame** by math.

Step 3: What transform: `translate()` really means

Let's say your image (glass in the center) is a layer texture.

Frame N (centered)

The GPU is told:

"Draw this rectangle at position (x=500, y=300)"

The texture is sampled and written there.

Frame N+1 (translated top-right)

The GPU is now told:

"Draw the **same rectangle** at position (x=650, y=150)"

That's it.

- Same texture
- Same pixels
- **Different vertex positions**

The GPU:

- Does **not** move anything
- Does **not** modify the texture
- Does **not** reuse framebuffer pixels

It simply draws the rectangle **somewhere else**.

Step 4: Why it *looks* like only the glass moved

Because:

- Background layer is drawn first
- Glass layer is drawn afterward
- Only the glass layer's rectangle changes position

Visually, it feels like:

"Only the glass moved"

But technically:

The entire scene was redrawn

Step 5: Visual mental model (very important)

✗ Wrong mental model

"There is an image already on screen and the GPU slides it upward"

✓ Correct mental model

"Each frame is a brand-new picture, reconstructed from layers using math"

Step 6: Mapping this to WebGL (to make it click)

If you've used WebGL:

```
glsl
// Vertex shader
gl_Position = projection * modelMatrix * vec4(position, 1.0);
```

- `modelMatrix` changes each frame
- `position` data stays the same
- Texture stays the same
- Framebuffer is rewritten

CSS transforms do **the same thing**, just hidden from you.

Why this is fast

- Changing a transform = updating **a few floats**
- Sampling a texture = extremely optimized

- No memory copy
- No CPU repaint

That's why it hits 60–120 FPS easily.

Final answer to your exact question

"Are you saying the image in framebuffer is the same?"

No.

The framebuffer is completely rewritten each frame.

"Are you saying the entire image is moved?"

No.

The image is **redrawn** at a different position using the same texture.

"Why does it look like only the glass moved?"

Because only the **glass layer's draw position changed**.

One-line summary (worth memorizing)

Transforms don't move pixels.

They change where the GPU draws the same pixels each frame.

If you want, next we can:

- draw an ASCII pipeline diagram
- compare `translate` vs `left/top`
- explain why scrolling is special

Just tell me 

so GPU does recreate the image in frame buffer every render

Yes — **that statement is correct**, with one very important precision.

The GPU recreates the *framebuffer* every frame, not the image (texture).

Let me lock this in cleanly so it finally "clicks".

What is recreated every render (frame)

✓ Framebuffer

- The framebuffer is the *final screen image*
- It is **redrawn every frame**
- This is true whether anything moved or not

So yes:

Every render → new framebuffer contents

What is NOT recreated

✗ Image / layer texture

- The texture containing your glass image:
 - Is created once (after paint)
 - Lives in GPU memory
 - Is reused across frames
 - It is **sampled**, not rebuilt
-

Exact pipeline for transform: `translate()`

Each frame:

1. Clear framebuffer
2. Draw background layer
3. Draw glass layer:
 - Same texture
 - New vertex positions (translated)
4. Write result into framebuffer

No pixels are moved.

No image is re-uploaded.

Only math + sampling.

Why this matters (performance)

Operation	Cost
Repainting image	✗ Expensive
Uploading texture	✗ Expensive
Updating transform matrix	✓ Very cheap

Operation	Cost
Redrawing framebuffer	✓ Normal GPU work

This is why:

- **transform / opacity** → smooth
 - **top/left/width** → jank
-

One sentence you can confidently say

"On each frame, the GPU re-renders the framebuffer by sampling existing textures with updated transforms."

That is **100% accurate**.

If you want, I can now explain:

- why 2D canvas behaves differently
- why **position: fixed** scrolling is special
- how browsers batch compositor layers

Just let me know.

difference between texture and frame buffer/

Great question — this is the *core* concept behind everything we discussed.
I'll explain it **conceptually first**, then map it to the browser.

1. Big picture (one sentence)

A **texture** is an input image stored on the GPU.

A **framebuffer** is the output image the GPU is currently drawing.

2. Texture

What it is

- A **read-only image** (for rendering purposes)
- Stored in **GPU memory**
- Used as **source data**

What it's used for

- Images ()
- Painted DOM layers
- Canvas contents
- Video frames

Key properties

- Sampled by shaders
- Not modified during rendering
- Can be reused across many frames

Analogy

Sticker

You can place the sticker anywhere, scale it, rotate it — but the sticker itself doesn't change.

3. Framebuffer

What it is

- The **render target**
- Represents the **current screen image**
- Written to every frame

What it's used for

- Final composited result
- What ends up on your monitor

Key properties

- Written pixel-by-pixel by the GPU
- Cleared and redrawn every frame
- Temporary — replaced continuously

Analogy

Whiteboard

You erase and redraw the whiteboard every frame.

4. How they work together (step by step)

When drawing an image:

1. GPU sets the framebuffer as the output
2. GPU draws a rectangle
3. For each pixel in that rectangle:
 - Sample color from the texture
 - Write color into the framebuffer

So:

```
arduino
Texture (read) —→ GPU —→ Framebuffer (write)
```

5. Why transforms don't need repaint

For `transform: translate()`:

- Texture → unchanged
- Transform matrix → changed
- Framebuffer → rewritten

Only math changes.

6. Browser-specific mapping

Browser concept	GPU concept
Painted DOM layer	Texture
Compositor layer	Textured quad
Screen	Framebuffer
Transform animation	Matrix update

7. When textures ARE recreated

Textures are recreated when:

- Text changes
- Layout/paint properties change
- Canvas 2D redraws

- Images load or resize

Then:

```
cpp
CPU paint → bitmap → upload → new texture
```

8. One-line memory trick

Textures are inputs.

Framebuffers are outputs.

If you want, next I can:

- explain offscreen framebuffers
- explain `filter: blur()` and why it's expensive
- map this to Vulkan / Metal terminology

Just say the word **(texture)**

so changing texture in GPU is expensive than frame buffer? when texture chanhing happens in gPU?

Yes — **you've got it exactly right.**

And your follow-up question ("when does texture changing happen?") is the right one to ask next.

1. Is changing a texture more expensive than drawing the framebuffer?

✓ Yes. Much more expensive.

Why:

Operation	Cost
Redrawing framebuffer	✓ Normal GPU work (fast)
Updating transform / opacity	✓ Very cheap
Sampling an existing texture	✓ Extremely optimized
Creating / updating a texture	✗ Expensive

The expensive part is uploading data:

- CPU → GPU transfer
- Memory allocation
- Pipeline synchronization

GPUs are built to **read textures**, not rewrite them constantly.

2. What does “changing a texture” mean?

A texture “change” means:

- New pixel data
- Or different size
- Or different content

This requires:

```
nginx
CPU bitmap → upload → GPU texture
```

That upload is the slow step.

3. When does texture changing happen in browsers?

Here are the **real, concrete cases** ↗

① DOM repaint (very common)

Any time the browser must repaint an element:

```
css
color
background
border
box-shadow
text
```

Pipeline:

```
cpp
Style → Layout → Paint → Upload new texture
```

Examples:

- Changing text content
- Hover effects that change background color

- Animating `box-shadow`
-

② Animating non-compositor properties

```
css
```

```
top  
left  
width  
height  
margin  
padding
```

Each frame:

- Layout recalculated
- Paint happens
- Texture updated each frame ✗

This is why these animations jank.

③ Canvas 2D redraw

```
js
```

```
ctx.clearRect(...)  
ctx.drawImage(...)
```

Every draw:

```
cpp
```

```
CPU rasterizes → upload → new texture
```

Canvas 2D is CPU-heavy.

④ WebGL texture updates

```
js
```

```
gl.texImage2D(...)  
gl.texSubImage2D(...)
```

You explicitly upload new pixel data.

Used for:

- Video textures
- Dynamic atlases
- Streaming data

⑤ Image or video frame changes

- Image loads
- Video advances frame

Each new frame = new texture data (or texture update)

4. When does texture NOT change?

Texture stays the same when:

- `transform`
- `opacity`
- `will-change: transform`
- Scrolling (often)

Only the **draw position** changes.

5. Why browsers work so hard to avoid texture changes

Because:

- Uploads stall the GPU pipeline
- Kill battery life
- Drop frames

That's why browsers:

- Promote elements to compositor layers
 - Cache textures
 - Encourage `transform` animations
-

6. Short rule you can rely on

If pixels change → texture changes → expensive

If only position/blending changes → framebuffer redraw → cheap

7. Final mental model (lock this in)

- Texture = *what*
- Transform = *where*
- Framebuffer = *result*

Change **what** → expensive

Change **where** → cheap

If you want, next I can:

- show how DevTools reveals texture uploads
- explain **filter vs opacity**
- explain why **will-change** can backfire

Just tell me 