FREX Overview

Demo Environment
MacBook Pro Intel 2.3 GHz 8-Core Intel Core i9
16GB RAM
AMD Radeon Pro 5500M 4 GB
MultiMC 5, Java 17
Minecraft 1.19.2
Fabric Loader 0.14.9
Fabric API 0.60.0

Demo Mods (partial list)
FREX 6.1.296
Canvas 1.0.2476
Render Bender 2.0.155
Exotic Blocks 2.0.224
Lithium 0.8.3
Phosphor 0.8.1
Ability 0.1.33 (unreleased)

Pipeline Shader
LumiLights v0.41 snapshot2
By spiralhalo
https://github.com/spiralhalo/LumiLights

Grondag September, 2022

Design Goal: Enable Amazing Content

- “Yes, you can do that.”
- Mods using the API are always compatible with each other
- Operate on any mod loader
- Avoid 3rd-party dependencies

Design Goal: Separate API and Implementation

- Provide a stable contract focused on intent
- Abstract OpenGL and Mojang classes
- Encourage 3rd-party implementations and useful extensions
- Support performant implementations with realistic lighting models
- Mods using the API work and look good with any implementation

MATERIALS

Making stuff glow

FREX `RenderMaterial`

- Attribute of a polygon
- Specifies intended visual effect
- Appearance depends on lighting model, etc. - but should always look right
- Independent of block/item/entity state - every quad in a model can be different
- Implementation is hidden

FREX `RenderMaterial`

- Three ways to obtain:
  - JSON material loader, reference by resource name
  - Java API: `MaterialFinder`
  - Register and retrieve as named resource via `MaterialManager`

Attaching Materials to Polygons

- JSON Material Maps
  - For resource pack makers or mods with simple (non-code) models
- Java API
  - Models that directly emit quads have fine-grained control
- JMX - JSON Model Extensions (different project)
  - Extends vanilla JSON models to attach materials directly
- 3rd-party model loaders and libraries can use Java API as JMX does

Fire!

JSON Material and Material Map

Material Definition
Unspecified properties default to solid terrain values
`assets/<modname>/materials`

emissive_no_diffuse_no_shadow.json

```
{
  "transform": true,
  "disableDiffuse": true,
  "emissive": true,
  "castShadows": false
}
```

Material Map
Applies named materials to selected quads for a block, block entity, item, fluid, entity or particle
`assets/<modname>/materialmaps/block`

fire.json

```
{
  "defaultMaterial": "frex:emissive_no_diffuse_no_shadow"
}
```

Wavy Grass

JSON Material and Material Map with Material Shader

Material Definition

low_foliage.json

```
{
  "vertexSource": "canvas:shaders/material/low_foliage.vert",
  "fragmentSource": "canvas:shaders/material/default.frag",
  "depthVertexSource": "canvas:shaders/material/low_foliage.vert",
  "disableAo": true,
  "disableDiffuse": true
}
```

Material Map

grass.json

```
{
  "defaultMaterial": "canvas:low_foliage"
}
```

Wavy Grass Material Shader

That grass ain't gonna wave itself

Vertex Shader GLSL

low_foliage.vert

```
#include frex:shaders/api/vertex.glsl
#include frex:shaders/api/world.glsl
#include frex:shaders/lib/math.glsl
#include frex:shaders/lib/noise/noise3d.glsl

/******************************************************
  canvas:shaders/material/low_foliage.vert

  Based on "GPU-Generated Procedural Wind Animations for Trees"
  by Renaldas Zioma in GPU Gems 3, 2007
  https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-6-gpu-generated-procedural-wind-animations-tr
******************************************************/

void frx_materialVertex() {
#ifdef ANIMATED_FOLIAGE
    float globalWind = 0.2 + frx_rainGradient * 0.2;
    float t = frx_renderSeconds * 0.05;
    // Azalea bush special case
    float texcoordy = (abs(frx_vertexNormal.y) > 0.9) ? 0.0 : frx_texcoord.y;
    float wind = snoise(vec3((frx_vertex.xz + frx_modelToWorld.xz) * 0.0625, t)) * (1.0 - texcoordy) * globalWind;

    frx_vertex.x += (cos(t) * cos(t * 3) * cos(t * 5) * cos(t * 7) + sin(t * 25)) * wind;
    frx_vertex.z += sin(t * 19) * wind;
#endif
}
```

Fragment Shader is empty (default) for wavy grass

Sunflower and Campfire

Material Map Selection Criteria

Sunflower: select based on block state variant

sunflower.json

```
{
  "defaultMaterial": "frex:standard",
  "variants": {
    "half=lower": {
      "defaultMaterial": "canvas:low_foliage"
    },
    "half=upper": {
      "defaultMaterial": "canvas:high_foliage"
    }
  }
}
```

Campfire: select based on sprite

campfire.json

```
{
  "defaultMap": {
    "spriteMap": [
      {
        "sprite": "minecraft:block/campfire_fire",
        "material": "frex:emissive_no_diffuse"
      },
      {
        "sprite": "minecraft:block/campfire_log_lit",
        "material": "canvas:warm_glow"
      }
    ]
  }
}
```

Redstone Ore

Per-Pixel Transformation with Material Shader

Material Definition

redstone.json

```
{
   "fragmentSource": "canvas:shaders/material/redstone.frag"
}
```

Material Map

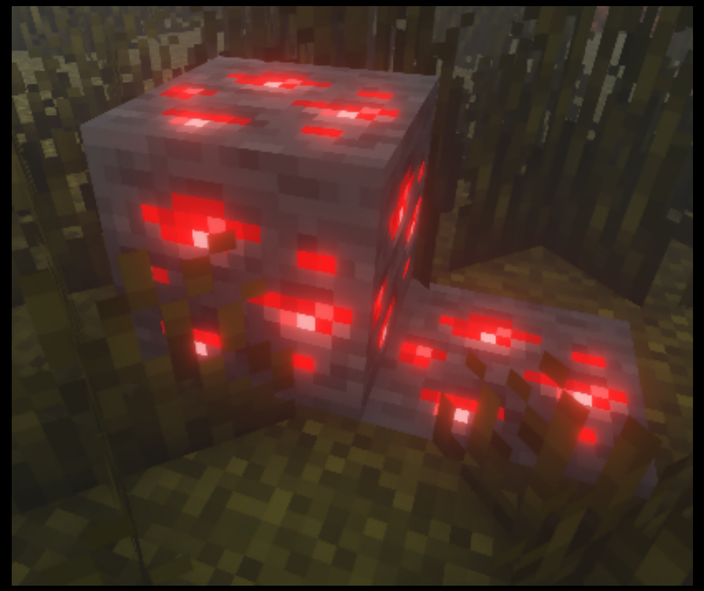
redstone_ore.json

```
{
   "defaultMaterial": "canvas:redstone"
}
```

Material Shader

redstone.frag

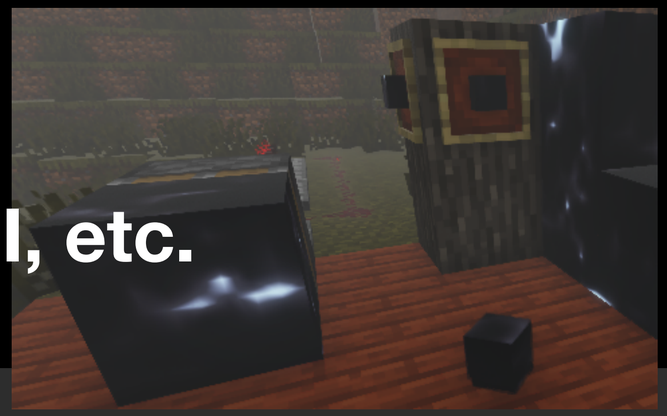
```
#include frex:shaders/api/fragment.glsl
#include frex:shaders/lib/math.glsl

/******************************************************
  canvas:shaders/material/redstone.frag
******************************************************/

void frx_materialFragment() {
#ifndef DEPTH_PASS
    float r = frx_smootherstep(0.1, 0.5, dot(frx_sampleColor.rgb, vec3(1.0, 0.0, -1.0)));
    float l = frx_smootherstep(0.8, 1.0, frx_luminance(frx_sampleColor.rgb));
    frx_fragEmissive = max(r, l);
#endif
}
```



- This approach tends to break with resource packs that have textures very different from vanilla
- Physically-Based Rendering (PBR) texture maps will be better in most use cases (covered later)

FX Shader Example

Same material works for block, entity, item, GUI, etc.



Material and Model Code

```java
final MeshFactory meshFactory = (meshBuilder, materialFinder, spriteFunc) -> {
    final RenderMaterial mat = materialFinder
            .shader(new ResourceLocation("renderbender", "shader/test.vert"), new ResourceLocation("renderbender", "shader/test.frag"))
            .castShadows(false)
            .find();

    final TextureAtlasSprite sprite = spriteFunc.getSprite("minecraft:block/gray_concrete");

    return meshBuilder.box(mat, -1, sprite, 0, 0, 0, 1, 1, 1).build();
};

StaticMeshModel.createAndRegisterProvider(b -> b.defaultParticleSprite("minecraft:block/gray_concrete"), meshFactory, "renderbender:shader");
```

test.vert

Material Vertex Shader

```
#include "frex:shaders/api/vertex.glsl"
#include "frex:shaders/lib/face.glsl"

/******************************************************
  renderbender:shader/test.vert
******************************************************/

void frx_materialVertex() {
#ifndef DEPTH_PASS
    frx_var0.xy = frx_faceUv(frx_vertex.xyz, frx_vertexNormal);
#endif
}
```

test.frag

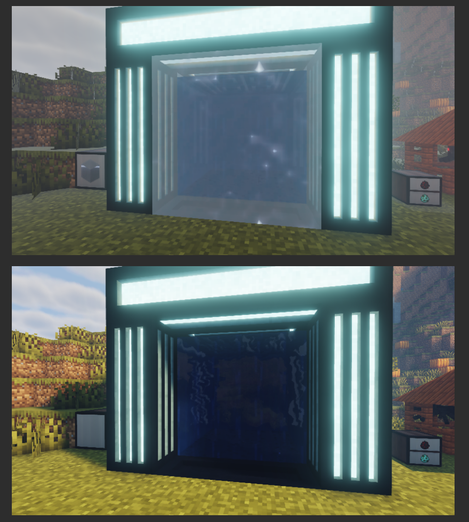
Material Fragment Shader

```
#include "frex:shaders/api/fragment.glsl"
#include "frex:shaders/lib/math.glsl"
#include "frex:shaders/api/world.glsl"

/******************************************************
  renderbender:shader/test.frag
******************************************************/

void frx_materialFragment() {
#ifndef DEPTH_PASS
    float t = frx_renderSeconds;
    float a = frx_noise2dt(frx_var0.xy * 2.0, t);
    float b = frx_noise2dt(frx_var0.xy * 4.0, t * 2.0);
    float c = frx_noise2dt(frx_var0.xy * 8.0, t * 4.0);
    float d = a * b * c;
    vec4 highlight = mix(vec4(1.0, 0.7, 1.0, 1.0), vec4(0.7, 1.0, 1.0, 1.0), d);
    float m = frx_smootherstep(0.0, 1.0, d);
    frx_fragColor = mix(frx_sampleColor, highlight, m);
    frx_fragEmissive = max(0, m - 0.5) * 2.0;
#endif
}
```

Conditional Materials

Material and Model Code

```java
public class Conditional {
    public static void initialize() {
        final MeshFactory meshFactory = (meshBuilder, materialFinder, spriteFunc) -> {
            final RenderMaterial mat = materialFinder
                    .shader(new ResourceLocation("renderbender", "shader/test.vert"), new ResourceLocation("renderbender", "shader/test.frag"))
                    .preset(MaterialConstants.PRESET_TRANSLUCENT)
                    .condition(condition.get())
                    .emissive(true)
                    .disableDiffuse(true)
                    .disableAo(true)
                    .find();

            final TextureAtlasSprite sprite = spriteFunc.getSprite("minecraft:block/snow");

            return meshBuilder.box(mat, 0x80DCEFFF, sprite, 0, 0, 0, 1, 1, 1).build();
        };

        StaticMeshModel.createAndRegisterProvider(b -> b.defaultParticleSprite("minecraft:block/snow"), meshFactory, "renderbender:conditional");
    }

    private static final Supplier<MaterialCondition> condition = Suppliers.memoize(() ->
        MaterialCondition.create(() -> {
            @SuppressWarnings("resource")
            final Entity entity = Minecraft.getInstance().cameraEntity;

            if (entity == null || entity.level == null) {
                return false;
            } else if (entity.level.isRaining()) {
                return true;
            } else if (entity instanceof final LivingEntity living) {
                return living.getMainHandItem().getItem() == ConditionalCommon.CONDITION_ITEM
                        || living.getOffhandItem().getItem() == ConditionalCommon.CONDITION_ITEM;
            } else {
                return false;
            }
        }));
}
```



Material Maps for Entities

Spider Material Map: Target by render type

`assets/<modname>/materialmaps/entity`

spider.json

```
{
    "map": [
        {
            "predicate": {
                "materialPredicate": {
                    "renderLayerName": "eyes"
                }
            },
            "material": "frex:emissive"
        }
    ]
}
```

Material Transform

- Alters existing material of polygon
- Avoids specifying complete material
- Useful across textures and targets
- Activated by “transform” attribute
- Don’t use for named materials

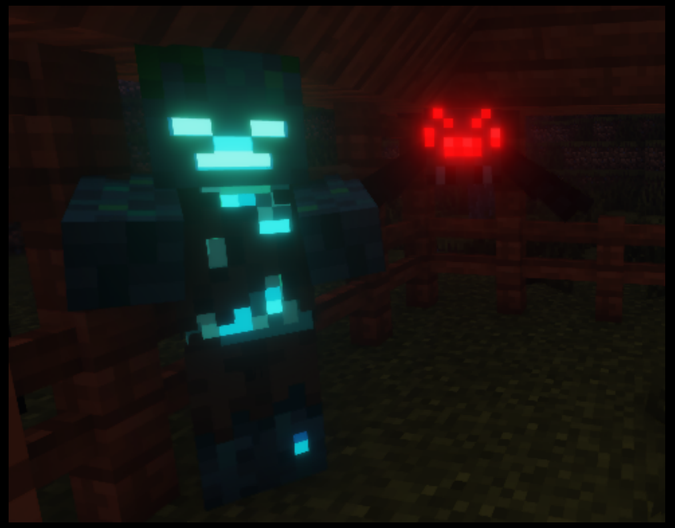
emissive.json

```
{
    "transform": true,
    "emissive": true
}
```

Drowned Material Map: Target by texture

drowned.json

```
{
    "map": [
        {
            "predicate": {
                "materialPredicate": {
                    "texture": "minecraft:textures/entity/zombie/drowned.png"
                }
            },
            "material": "frex:emissive"
        }
    ]
}
```



Material Maps for Other Game Objects

Block Entity: Beacon

`assets/<modname>/materialmaps/block_entity`

beacon.json

```
{
    "map": [
        {
            "predicate": {
                "materialPredicate": {
                    "renderLayerName": "beacon_beam"
                }
            },
            "material": "frex:emissive_no_shadow"
        }
    ]
}
```

Particle: Nether Portal

`Assets/<modname>/materialmaps/particle`

portal.json

```
{
    "material": "frex:emissive_no_diffuse"
}
```

Fluid: Lumi Lights Water

`Assets/<modname>/materialmaps/fluid`

water.json

```
{
  "defaultMaterial": "lumi:tasty_liquid"
}
```

Item: Acacia Sapling

Most common use is preventing animation of blocks item

`Assets/<modname>/materialmaps/item`

acacia_sapling.json

```
{
}
```

FREX RenderMaterial vs Mojang RenderType

RenderMaterial

- Stable contract
- Specifies intent, implementation is flexible
- Immutable and queryable, can copy to finder for modification
- Fabulous mode works with more use cases (implementation-dependent)
- Namespaced material registry - can share materials across mods
- JSON loaders, material maps, etc.

RenderType

- Mojang can change at any time
- Specifies implementation directly, difficult to optimize or extend
- Immutable, not easily queryable
- Can still serve as implementation!
- FREX RenderTypeUtil can convert most vanilla RenderTypes to RenderMaterial

MaterialFinder Methods

```
MaterialFinder preset(int preset);
MaterialFinder disableColorIndex(boolean disable);
MaterialFinder disableDiffuse(boolean disable);
MaterialFinder disableAo(boolean disable);
MaterialFinder emissive(boolean isEmissive);
MaterialFinder unlit(boolean unlit);
MaterialFinder blur(boolean blur);
MaterialFinder condition(@Nullable MaterialCondition condition);
MaterialFinder cull(boolean cull);
MaterialFinder cutout(int cutout);
MaterialFinder decal(int decal);
MaterialFinder depthTest(int depthTest);
MaterialFinder discardsTexture(boolean discardsTexture);
MaterialFinder flashOverlay(boolean flashOverlay);
MaterialFinder foilOverlay(boolean foilOverlay);
MaterialFinder overlay(int u, int v);
MaterialFinder fog(boolean enable);
MaterialFinder hurtOverlay(boolean hurtOverlay);
MaterialFinder lines(boolean lines);
MaterialFinder shader(MaterialShader shader);
MaterialFinder sorted(boolean sorted);
MaterialFinder target(int target);
MaterialFinder texture(MaterialTexture texture);
MaterialFinder transparency(int transparency);
MaterialFinder unmipped(boolean unmipped);
MaterialFinder writeMask(int writeMask);
MaterialFinder castShadows(boolean castShadows);
MaterialFinder label(String name);

MaterialFinder clear();
MaterialFinder copyFrom(RenderMaterial material);
```

SHADER API

Material Shaders vs Pipeline Shaders

`assets/<modname>/shaders/material`

- Attribute of a material
- One per polygon
- Provided by content makers as part of mods or resource packs
- Handles VFX for specific content
- Can have many material shaders from different mods in same scene
- Work with any pipeline

`assets/<modname>/shaders/pipeline`

- Modular part of GPU renderer implementation
- Analogous to OF “shader pack”
- Provided by pipeline authors
- Handles lighting, application of material shaders, shadows and other global effects
- Selected/configured by player
- Only one can be active

Vertex and Fragment shaders distinguished by naming convention (.vert .frag)

Renderer vs Pipeline

Renderer

- Implements Java API
- Implements most of GLSL API
- GL state management
- Resource loading
- Shader compilation
- Pipeline configuration
- Debugging facilities

Pipeline Shader

- Implements some of GLSL API
- Depends on renderer for buffering, encoding, state management
- Handles lighting, application of material shaders, shadows and other global effects
- Default pipelines ship with renderer
- 3rd-party pipelines selected by player. Goal is cross-renderer compatibility.

GLSL API Hides Implementation

- Header declarations are symbolic tokens that convey data type and mutability
- Token implementation will vary by renderer
- For example, a float attribute in API could be implemented as…
  - Uniform
  - Input variable
  - Global variable
  - Component of a float vector
  - Component of a struct
- `const` keyword in header indicates immutability but is not supported in shaders

Implementation Methods

```
/*
* Called by renderer after all variables are initialized and before
* frx_pipelineVertex() is called to update vertex outputs.
*
* The running vertex shader will have multiple, renamed versions of
* this method - one for each unique material vertex shader present
* in the game. The specific method called is controlled by the material
* associated with the current triangle.  If the current material does
* not define a custom material shader, no extra processing will happen
* before frx_pipelineVertex() is called.
*/
void frx_materialVertex();

/*
 * Called by renderer after frx_materialVertex() completes.
 * Pipeline authors implement this method to read vertex data,
 * apply transformation and update as needed. Pipeline vertex shaders
 * may also capture additional vertex outputs as needed.
 * These steps are specific to the design of each pipeline.
 *
 * The pipeline shader is responsible for ALL updates to vertex outputs,
 * including those defined by the FREX API.
 */
void frx_pipelineVertex();
```

```
/*
 * Called by renderer after all variables are initialized and before
 * frx_pipelineFragment() is called to output a fragment.
 *
 * The running fragment shader will have multiple, renamed versions of
 * this method - one for each unique material fragment shader present
 * in the game. The specific method called is controlled by the material
 * associated with the current triangle.  If the current material does
 * not define a custom material shader, no extra processing will happen
 * before frx_pipelineFragment() is called.
 */
void frx_materialFragment();

/*
 * Called by renderer after frx_materialFragment() completes.
 * Pipeline authors implement this method to read fragment data,
 * apply transformation or shading and output to one or more framebuffer
 * attachments.  These steps are specific to the design of each pipeline.
 *
 * The pipeline shader is responsible for ALL WRITES.
 * The renderer will not update depth or any color attachment.
 */
void frx_pipelineFragment();
```

Shader API Specification

- Specification ships as .h files in assets/frex/shaders/api
- Renderers provide .glsl implementation files for every header file in spec

header.h	supports conditional compilation based on high-level state
fog.h	fog parameters for the current primitive
material.h	material properties of current primitive
player.h	player information
sampler.h	direct sampler access and related utilities
view.h	coordinate spaces, camera and transformations
world.h	world information
vertex.h fragment.h	methods and attributes implemented/used by material/pipeline shaders

Shader API location in FREX repo is https://github.com/vram-guild/frex/tree/1.19/common/src/main/resources/assets/frex/shaders

Pipeline Example: Canvas Standard

standard.vert

```
#include canvas:shaders/pipeline/diffuse.glsl
#include frex:shaders/api/view.glsl
#include frex:shaders/api/player.glsl
#include canvas:handheld_light_config

/******************************************************
  canvas:shaders/pipeline/standard.vert
******************************************************/

#if HANDHELD_LIGHT_RADIUS != 0
flat out float _cvInnerAngle;
flat out float _cvOuterAngle;
out vec4 _cvViewVertex;
#endif

void frx_pipelineVertex() {
    if (frx_isGui) {
        gl_Position = frx_guiViewProjectionMatrix * frx_vertex;
        frx_distance = length(gl_Position.xyz);
#if HANDHELD_LIGHT_RADIUS != 0
        _cvViewVertex = gl_Position;
#endif
    } else {
        frx_vertex += frx_modelToCamera;
        vec4 viewCoord = frx_viewMatrix * frx_vertex;
        frx_distance = length(viewCoord.xyz);
        gl_Position = frx_projectionMatrix * viewCoord;
#if HANDHELD_LIGHT_RADIUS != 0
        _cvViewVertex = viewCoord;
#endif
    }

#if HANDHELD_LIGHT_RADIUS != 0
    _cvInnerAngle = sin(frx_heldLightInnerRadius);
    _cvOuterAngle = sin(frx_heldLightOuterRadius);
#endif

#if DIFFUSE_SHADING_MODE != DIFFUSE_MODE_NONE
    pv_diffuse = p_diffuse(frx_vertexNormal);
#endif
}
```

standard.frag

```
#include canvas:shaders/pipeline/standard_header.frag
#include canvas:shaders/pipeline/common.frag

/******************************************************
  canvas:shaders/pipeline/standard.frag
******************************************************/

void frx_pipelineFragment() {
    vec4 baseColor = p_writeBaseColorAndDepth();
    fragColor[TARGET_EMISSIVE] = vec4(frx_fragEmissive * baseColor.a, 0.0, 0.0, 1.0);
}
```

Shader Distribution and Dependencies

- FREX shaders are resources
- Renderers and pipelines should maintain separate namespaces - don't use `frex` except for API implementations
- `#include` directive
  - References specific files in resource tree (include file extension)
  - Implemented by renderer
  - Supports nesting
- FREX includes a small GLSL library to ease development, avoid duplication
- Mods/packs can include their own GLSL libraries or depend on others but must ensure such dependencies are part of distribution

Versions and Extensions

- Renderer implementations must support GLSL 330 and should ship with at least one 330-compatible pipeline shader (which should be the default).
- Renderer implementations may use OGL/GLSL extensions which are reliably available on hardware supported by Mojang.
- Material shaders should stay within GLSL 330 features.
- Renderers can provide support for pipeline shaders to use GLSL features > 330 for advanced shading models. Such pipelines should be clearly labeled as incompatible on Macs. Primary use case is compute shaders.

MODELS

Making the stuff that glows

Model Outputs

Where do polygons go?

QuadEmitter

- Random access to all vertices, explicitly emit quad when done
- Can load from BakedQuad
- Ideal for block/item models

VertexEmitter

- Stream individual vertices, polygon completion is implicit
- Extends Mojang VertexConsumer
- Allows material specification
- Meant for entities, particles and other code that already depends on VertexConsumer

Surprise Reveal: They Same

QuadEmitter and VertexEmitter extend QuadSink

```
public interface QuadSink {
    QuadEmitter asQuadEmitter();

    VertexEmitter asVertexEmitter();

    PooledQuadEmitter withTransformQuad(InputContext context, QuadTransform tr

    PooledVertexEmitter withTransformVertex(InputContext context, QuadTransfor

    /** Has no effect for non-pooled emitters. */
    default void close() {
        // NOOP
    }

    /**
     * Will be true for instances returned from {@link #withTransformQuad(Inpu
     * or {@link #withTransformVertex(InputContext, QuadTransform)}.  Primary
     * if for renderers to avoid early culling tests when a transform is prese
     *
     * @return {@code true} when this instance applies a {@link QuadTransform}
     */
    default boolean isTransformer() {
        return false;
    }
}
```

- API guarantees one is always convertible to the other
- Use whichever pattern fits your use case
- Good for rendering entity “models” in terrain or block/item models as entities
- Reasonable limitations apply:
  - Don’t convert in the middle of quad
  - QuadEmitter is for quads…

QuadEmitter Highlights

```
QuadEmitter material(RenderMaterial material);
QuadEmitter defaultMaterial(RenderMaterial material);
QuadEmitter cullFace(@Nullable Direction face);
QuadEmitter nominalFace(Direction face);
QuadEmitter colorIndex(int colorIndex);
QuadEmitter fromVanilla(int[] quadData, int startIndex);
QuadEmitter fromVanilla(BakedQuad quad, RenderMaterial material, Direction cullFace);
QuadEmitter tag(int tag);
QuadEmitter pos(int vertexIndex, float x, float y, float z);
QuadEmitter normal(int vertexIndex, float x, float y, float z);
QuadEmitter tangent(int vertexIndex, float x, float y, float z);
QuadEmitter lightmap(int vertexIndex, int lightmap);
QuadEmitter vertexColor(int vertexIndex, int color);
QuadEmitter emit();

/** * Set texture coordinates relative to the "raw" texture.
 * For sprite atlas textures, the coordinates must be
 * relative to the atlas texture, not relative to the sprite.
 */
QuadEmitter uv(int vertexIndex, float u, float v);

/**
 * Sets texture coordinates relative to the given texture sprite.
 * For sprite atlas textures, this may be more convenient than
 * interpolating the coordinates directly.  Additionally, this method
 * will often be more performant for renderers that need to track
 * sprite identity within meshes or that use sprite-relative coordinates
 * in shaders.
 *
 * <p>Note that the material for this quad must be associated with a
 * sprite atlas texture, and the sprite must belong to that atlas texture.
 * If this condition is unmet, or if the sprite is null, will operate
 * as if {@link #uv(int, float, float)} had been called for all four vertices.
 */
QuadEmitter uvSprite(@Nullable TextureAtlasSprite sprite, float u0, float v0, float u1,

/**
 * Assigns sprite atlas u,v coordinates to this quad for the given sprite.
 * Can handle UV locking, rotation, interpolation, etc. Control this behavior
 * by passing additive combinations of the BAKE_ flags defined in this interface.
 * Behavior for {@code spriteIndex > 0} is currently undefined.
 */
QuadEmitter spriteBake(TextureAtlasSprite sprite, int bakeFlags);
```

- Tag is an arbitrary value mods can use for later inspection/ transformation - doesn’t go to GPU
- Tangent is for PBR support. Vertex normals and tangents are auto-computed if not provided
- Accepts UV coordinates relative to sprite (0-1) instead of baked atlas coordinates
- Handles most texture baking use cases with minimal effort
- Also implements QuadView for inspection and conversion to BakedQuad

VertexEmitter Extensions

```java
public interface VertexEmitter extends VertexConsumer, QuadSink {
    VertexEmitter defaultMaterial(RenderMaterial material);

    /**
     * Sets state to be included with normals and material if they are included.  Call once
     * whenever material changes, including default state or revert
     * to default state of the render state.
     */
    VertexEmitter material(RenderMaterial material);

    VertexEmitter vertex(float x, float y, float z);

    /**
     * @param color rgba - alpha is high byte, red and blue pre-swapped if needed
     */
    VertexEmitter color(int color);
```

- Mostly vanilla except for RenderMaterial methods
- Vertex and color added to reduce call/instantiation overhead, eliminate boilerplate code
- Tangent should be added for PBR support

FREX Models are Functions

```java
@FunctionalInterface
public interface DynamicModel {
    void renderDynamic(InputContext input, QuadSink output);
}
```

```java
@FunctionalInterface
public interface BlockModel extends DynamicModel {
    void renderAsBlock(BlockInputContext input, QuadSink output);
```

```java
@FunctionalInterface
public interface ItemModel extends DynamicModel {
    void renderAsItem(ItemInputContext input, QuadSink output);
```

```java
@FunctionalInterface
public interface EntityModel<T extends Entity> extends DynamicModel {
    void renderAsEntity(EntityInputContext<T> input, QuadSink output);
```

- Model interfaces are meant to be mixed. (Implement whichever methods apply.)
- Entity and similar model types partially specified, not yet complete

Implementing FREX Models

- Vanilla models still work
  - JSON Materials / MaterialMaps can handle many use cases
  - FREX makes BakedModel implement BlockModel and ItemModel
- Use JMX or 3rd-party model libraries
- Use Java API (“code” models)

```
@Mixin(BakedModel.class)
public interface MixinBakedModel extends BlockItemModel {
    @Override
    default void renderAsItem(ItemInputContext input, QuadSink output) {
        BakedModelTranscoder.accept((BakedModel) this, input, output.asQuadEmitter());
    }

    @Override
    default void renderAsBlock(BlockInputContext input, QuadSink output) {
        BakedModelTranscoder.accept((BakedModel) this, input, output.asQuadEmitter());
    }
}
```

Code Model Approaches

- Static, pre-built meshes (fastest)
- Cached dynamic meshes (fast, can be complicated)
- Transform pre-built meshes (still fast, depending on transform)
- Transform vanilla models (not as efficient as meshes, but OK)
- Direct output to QuadSink (slowest but usable when needed)
- Hybrid approaches / composite models are fine

Static Mesh Model

```java
private static void registerRoundModel(String blockPath, boolean smoothNormals, Function<MaterialFinder, RenderMaterial> materialFunc) {
    final MeshFactory meshFactory = (meshBuilder, materialFinder, spriteFunc) -> {
        final TextureAtlasSprite sprite = spriteFunc.getSprite("minecraft:block/white_concrete");
        makeIcosahedron(new Vector3f(0.5f, 0.5f, 0.5f), 0.5f, meshBuilder.getEmitter(), materialFunc.apply(materialFinder), sprite, smoothNormals);
        return meshBuilder.build();
    };

    ModelProviderRegistry.registerBlockItemProvider(StaticMeshModel.createProvider(b -> b.defaultParticleSprite("minecraft:block/white_concrete"), meshFactory), blockPath);
}

private static void makeIcosahedron(Vector3f center, float radius, QuadEmitter qe, RenderMaterial material, TextureAtlasSprite sprite, boolean smoothNormals) {
    final Vector3f[] vertexes = new Vector3f[12];
    Vector3f[] normals = new Vector3f[12];
    // create 12 vertices of a icosahedron

    // ...MATH GOES HERE... OMITTED FOR BREVITY...

    // create 20 triangles of the icosahedron
    makeIcosahedronFace(true, 0, 11, 5, vertexes, normals, qe, material, sprite);
    // ETC. ETC. 19X more
}


private static void makeIcosahedronFace(boolean topHalf, int p1, int p2, int p3, Vector3f[] points, Vector3f[] normals, QuadEmitter qe, RenderMaterial material, TextureAtlasSpri
    if (topHalf) {
        qe.pos(0, points[p1]).uv(0, 1, 1).vertexColor(-1, -1, -1, -1);
        qe.pos(1, points[p2]).uv(1, 0, 1).vertexColor(-1, -1, -1, -1);
        qe.pos(2, points[p3]).uv(2, 1, 0).vertexColor(-1, -1, -1, -1);
        qe.pos(3, points[p3]).uv(3, 1, 0).vertexColor(-1, -1, -1, -1);
    } else {
        qe.pos(0, points[p1]).uv(0, 0, 0).vertexColor(-1, -1, -1, -1);
        qe.pos(1, points[p2]).uv(1, 1, 0).vertexColor(-1, -1, -1, -1);
        qe.pos(2, points[p3]).uv(2, 0, 1).vertexColor(-1, -1, -1, -1);
        qe.pos(3, points[p3]).uv(3, 0, 1).vertexColor(-1, -1, -1, -1);
    }

    if (normals != null) {
        qe.normal(0, normals[p1]);
        qe.normal(1, normals[p2]);
        qe.normal(2, normals[p3]);
        qe.normal(3, normals[p3]);
    }

    qe.spriteBake(sprite, QuadEmitter.BAKE_NORMALIZED);
    qe.material(material);
    qe.emit();
}
```

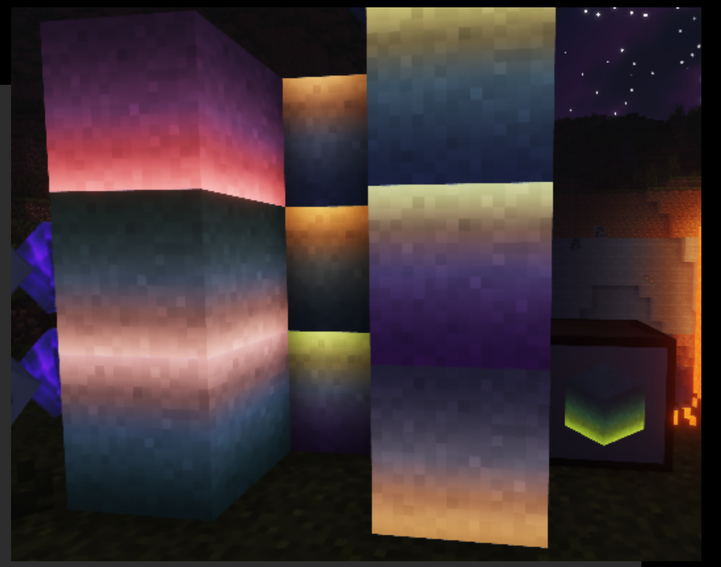
Mesh

```java
/**
 * A bundle of one or more {@link QuadView} instances encoded by the renderer,
 * typically via {@link Renderer#meshBuilder()}.
 *
 * <p>Similar in purpose to the {@code List<BakedQuad>} instances returned by BakedModel, but
 * affords the renderer the ability to optimize the format for performance
 * and memory allocation.
 *
 * <p>Only the renderer should implement or extend this interface.
 */
@FunctionalInterface
public interface Mesh {
    /**
     * Use to access all of the quads encoded in this mesh. The quad instances
     * sent to the consumer will likely be threadlocal/reused and should never
     * be retained by the consumer.
     */
    void forEach(Consumer<QuadView> consumer);

    default void outputTo(QuadEmitter editor) {
        forEach(v -> {
            v.copyTo(editor);
            editor.emit();
        });
    }

    default void outputTo(QuadSink quadSink) {
        outputTo(quadSink.asQuadEmitter());
    }

    Mesh EMPTY = c -> { };
}
```

MeshBuilder

```java
/**
 * Similar in purpose to {@link BufferBuilder} but simpler
 * and not tied to NIO or any other specific implementation,
 * plus designed to handle both static and dynamic building.
 *
 * <p>Decouples models from the vertex format(s) used by
 * ModelRenderer to allow compatibility across diverse implementations.
 */
public interface MeshBuilder {
    /**
     * Returns the {@link QuadEmitter} used to append quad to this mesh.
     * Calling this method a second time invalidates any prior result.
     * Do not retain references outside the context of building the mesh.
     */
    QuadEmitter getEmitter();

    /**
     * Returns a new {@link Mesh} instance containing all
     * quads added to this builder and resets the builder to an empty state.
     */
    Mesh build();
```

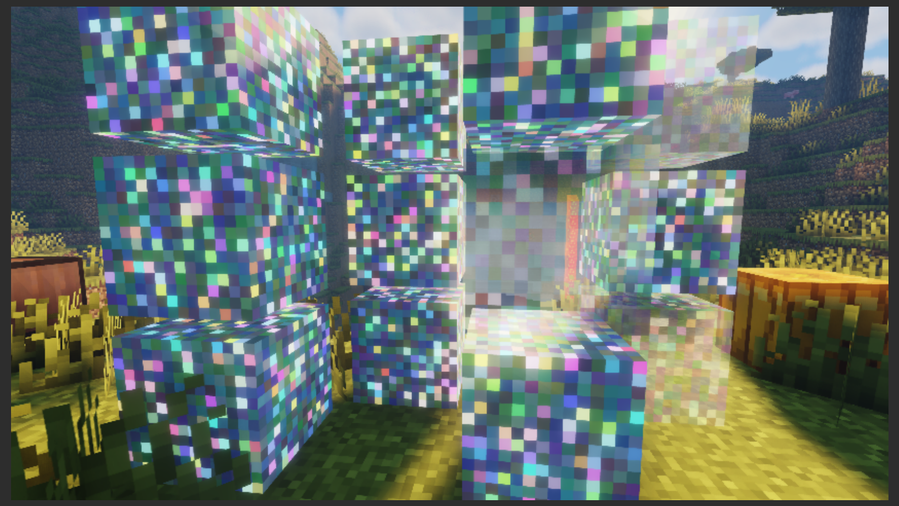
Transform Vanilla Model



```
35          final QuadTransform transform = (ctx, in, out) -> {
36              final var random = ctx.random();
37              final int topColor = DynamicGlow.randomPastelColor(random);
38              final int bottomColor = DynamicGlow.randomPastelColor(random);
39              final boolean topGlow = random.nextBoolean();
40              final int topLight = topGlow ? ColorUtil.FULL_BRIGHTNESS : 0;
41              final int bottomLight = topGlow ? 0 : ColorUtil.FULL_BRIGHTNESS;
42              in.copyTo(out);
43
44              for (int i = 0; i < 4; i++) {
45                  if (Mth.equal(out.y(i), 0)) {
46                      out.vertexColor(i, bottomColor).lightmap(i, bottomLight);
47                  } else {
48                      out.vertexColor(i, topColor).lightmap(i, topLight);
49                  }
50              }
51
52              out.emit();
53          };
54
55          TransformingModel.createAndRegisterProvider(
56              b -> b.defaultParticleSprite("minecraft:block/white_concrete_powder"),
57              () -> (BlockItemModel) BlockModel.get(Blocks.WHITE_CONCRETE_POWDER.defaultBlockState()),
58              transform,
59              "renderbender:dynamic_glow");
```

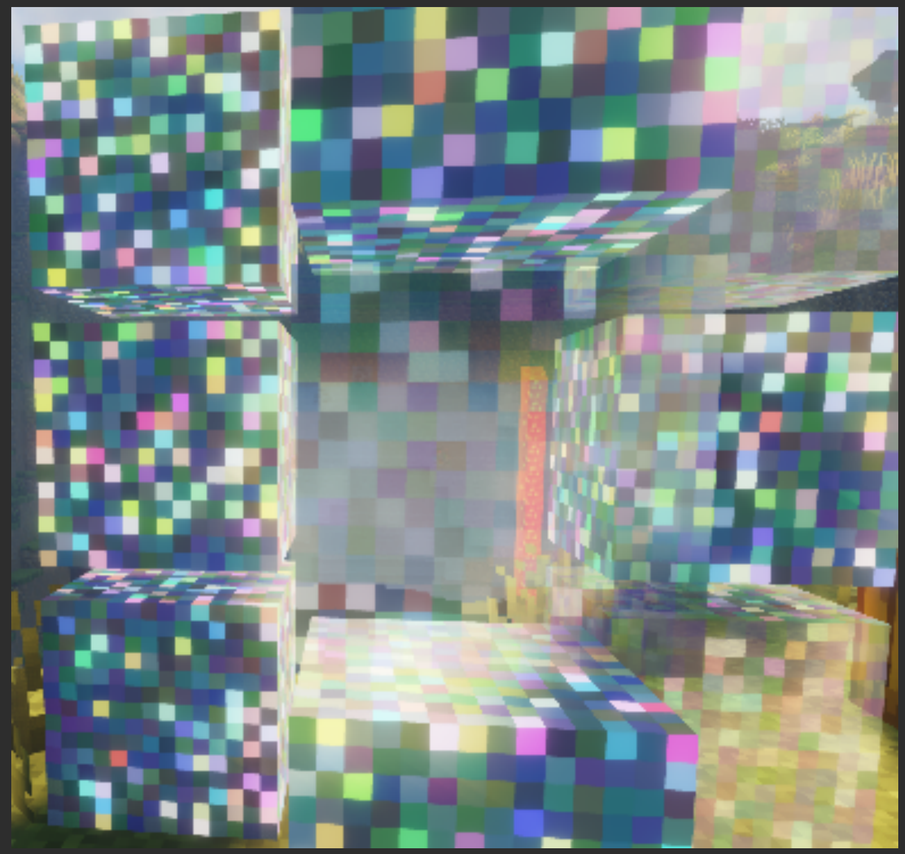
Transform Static Mesh - Mesh Part

```
final MeshFactory meshFactory = (meshBuilder, materialFinder, spriteFunc) -> {
    final TextureAtlasSprite sprite = spriteFunc.getSprite("minecraft:block/white_concrete");
    final RenderMaterial mat = materialFinder.find();
    final float PIXEL = 1f/16f;
    final QuadEmitter qe = meshBuilder.getEmitter();
    int t = 0;

    for (int d = 0; d < 6; d++) {
        final Direction face = Direction.from3DDataValue(d);

        for (int i = 0; i < 14; i++) {
            final float u = PIXEL + PIXEL * i;

            for (int j = 0; j < 14; j++) {
                final float v = PIXEL + PIXEL * j;
                qe.tag(t++);
                qe.material(mat).square(face, u, v, u + PIXEL, v + PIXEL, PIXEL)
                .vertexColor(-1, -1, -1, -1)
                .spriteBake(sprite, QuadEmitter.BAKE_LOCK_UV).emit();
            }
        }
    }

    return meshBuilder.build();
};

TransformingMeshModel.createAndRegisterProvider(b -> b.defaultParticleSprite("minecraft:block/white_concrete_powder"),
        meshFactory, transform, "renderbender:be_data");
```



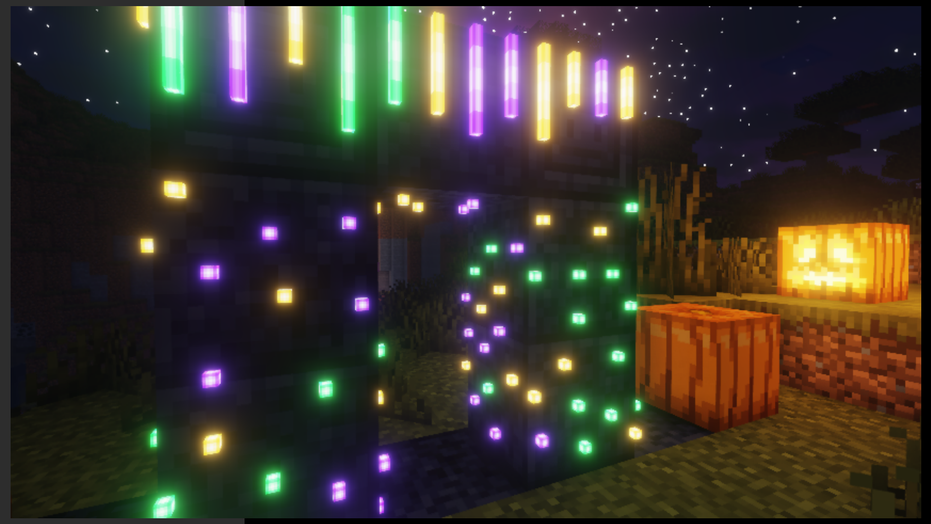
Transform Static Mesh - Transform Part

```
BlockEntityRenderData.registerProvider(BlockEntityDataCommon.BE_TEST_TYPE, be -> ((BeTestBlockEntity) be).getRenderData());

final var finder = MaterialFinder.threadLocal();
final var matSolid = finder.clear().preset(MaterialConstants.PRESET_SOLID).find();
final var matSolidGlow = finder.clear().preset(MaterialConstants.PRESET_SOLID).disableDiffuse(true).emissive(true).disableAo(true).find();
final var matTrans = finder.clear().preset(MaterialConstants.PRESET_TRANSLUCENT).find();
final var matTransGlow = finder.clear().preset(MaterialConstants.PRESET_TRANSLUCENT).disableDiffuse(true).emissive(true).disableAo(true).find();

final QuadTransform transform = (ctx, in, out) -> {
    final RenderMaterial mat;
    final RenderMaterial matGlow;
    final boolean translucent;
    int[] data = null;

    if (ctx.type() == Type.BLOCK) {
        final var bctx = (BlockInputContext) ctx;
        data = (int[]) bctx.blockEntityRenderData(bctx.pos());
    }

    if (data == null) {
        mat = matSolid;
        matGlow = matSolidGlow;
        translucent = false;
        data = BeTestBlockEntity.ITEM_COLORS;
    } else {
        if (ctx.random().nextInt(4) == 0) {
            mat = matTrans;
            matGlow = matTransGlow;
            translucent = true;
        } else {
            mat = matSolid;
            matGlow = matSolidGlow;
            translucent = false;
        }
    }

    in.copyTo(out);
    final int s = data == null ? -1 : data[out.tag()];
    final int c = translucent ? 0x80000000 | (0xFFFFFF & s) : s;
    out.material((s & 0x3) == 0 ? matGlow : mat).vertexColor(c, c, c, c);
    out.emit();
};
```



Direct Output Model (Excerpts)

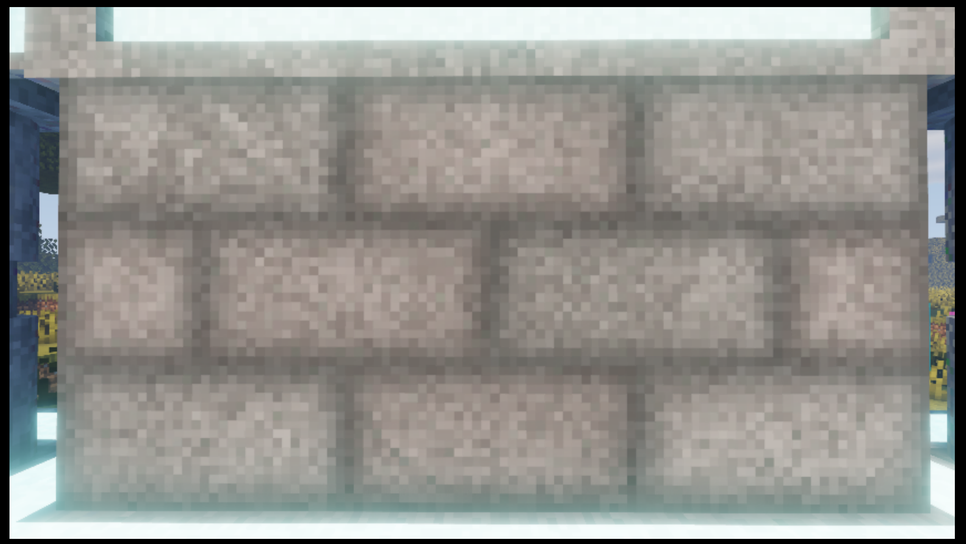
```
protected static final RenderMaterial material = MaterialFinder.threadLocal().emissive(true).disableAo(true)
        .disableDiffuse(true).preset(MaterialConstants.PRESET_SOLID).find();

static final BlockModel BLOCK_MODEL = (in, out) -> {
    final var state = in.blockState();

    if (!(state.getBlock() instanceof FestiveLightsBlock)) {
        return;
    }

    final FestiveLightsBlock block = (FestiveLightsBlock) state.getBlock();

    final int[] colors = block.colors;

    final QuadEmitter qe = out.asQuadEmitter();
    final RandomSource rand = in.random();
    final TextureAtlasSprite sprite = SpriteProvider.forBlocksAndItems().getSprite(SPRITE_ID);

    if (block.isPendant()) {
```



```
protected static void emitQuads(TextureAtlasSprite sprite, QuadEmitter qe, int color, float x, float y, float z, float step) {
    emitFace(sprite, qe, Direction.UP, color, x, 1 - z - step, 1 - y - step, step);
    emitFace(sprite, qe, Direction.DOWN, color, x, z, y, step);
    emitFace(sprite, qe, Direction.EAST, color, 1 - z - step, y, 1 - x - step, step);
    emitFace(sprite, qe, Direction.WEST, color, z, y, x, step);
    emitFace(sprite, qe, Direction.NORTH, color, 1 - x - step, y, z, step);
    emitFace(sprite, qe, Direction.SOUTH, color, x, y, 1 - z - step, step);
}

protected static void emitFace(TextureAtlasSprite sprite, QuadEmitter qe, Direction face, int color, float x, float y, float d, float step) {
    final float u = COLOR_UV[color * 2];
    final float v = COLOR_UV[color * 2 + 1];

    qe
        .material(material)
        .square(face, x, y, x + step, y + step, d)
        .uv(0, u, v)
        .uv(1, u + UV_STEP, v)
        .uv(2, u + UV_STEP, v + UV_STEP)
        .uv(3, u, v + UV_STEP)
        .vertexColor(-1, -1, -1, -1)
        .spriteBake(sprite, QuadEmitter.BAKE_NORMALIZED);
    qe.emit();
}
```

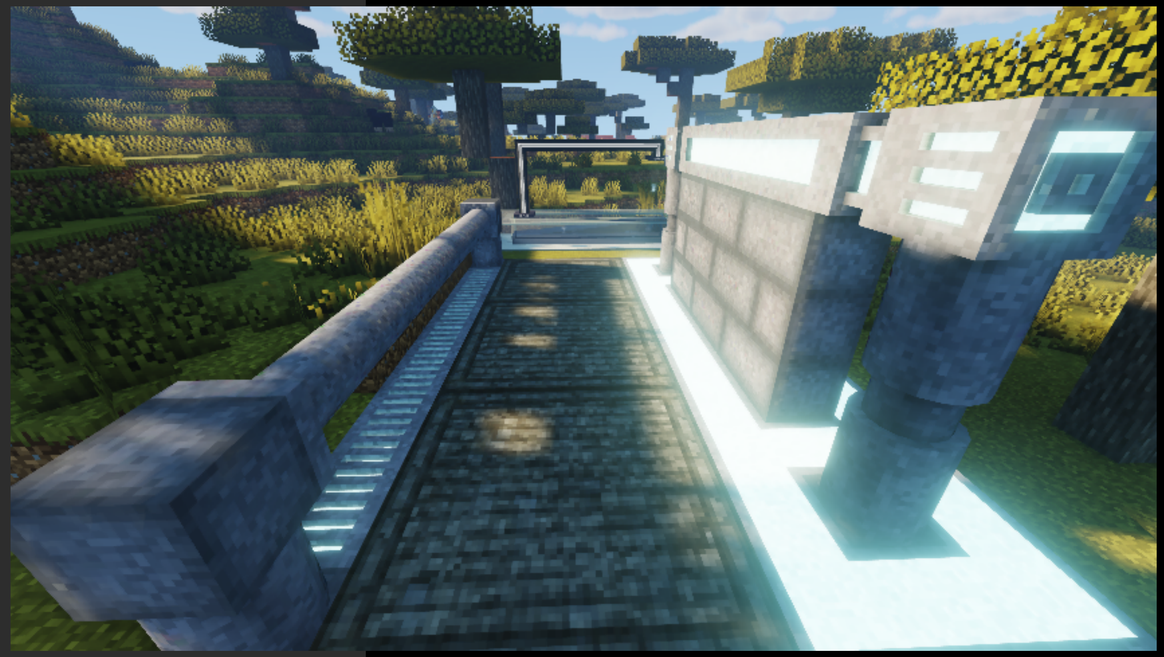
Cached Dynamic Mesh - Model Definition (Exotic Matter library)

```
final XmPaint connectedPaint = XmPaint.finder()
        .textureDepth(3)
        .texture(0, CoreTextures.BIGTEX_SANDSTONE)
        .vertexProcessor(0, VertexProcessors.SPECIES_VARIATION)
        .textureColor(0, 0xFF9090A0)
        .texture(1, XmTextures.TILE_NOISE_BLUE_A)
        .textureColor(1, 0xA0656d75)
        .texture(2, CoreTextures.BORDER_SMOOTH_BLEND)
        .textureColor(2, 0x802b2f33)
        .find();

final Block fancyStone = Xb.block(BlockNames.BLOCK_FANCY_STONE, new Block(Block.Properties.copy(Blocks.STONE)));
XmBlockRegistry.addBlock(fancyStone, PrimitiveStateFunction.ofDefaultState(
        Cube.INSTANCE.newState()
        .paintAll(mainPaint)
        .releaseToImmutable()));

SpeciesBlock.species(fancyStone, BlockNames.BLOCK_CONNECTED_FANCY_STONE, connectedPaint);
```



Cached Dynamic Mesh - Model Implementation (Excerpts)

```java
@Environment(EnvType.CLIENT)
private Mesh mesh = null;

@Environment(EnvType.CLIENT)
public Mesh mesh() {
    Mesh result = mesh;

    if (result == null) {
        result = PaintManager.paint(this);
        mesh = result;
    }

    return result;
}

@Override
@Environment(EnvType.CLIENT)
public void renderAsBlock(BlockInputContext input, QuadSink output) {
    primitive.emitBlockMesh(mesh(), input, output);
}
```



```java
@Environment(EnvType.CLIENT)
default void emitBlockMesh(Mesh mesh, BlockInputContext input, QuadSink output) {
    mesh.outputTo(output);
}
```

Fluid Models

```java
/**
 * Identical in operation to {@link BlockModel} but for fluids.
 *
 * <p>A FREX-compliant renderer will call this - in addition to the block quad emitter - for
 * block state with a non-empty fluid state.  Block state is passed instead of fluid state
 * to keep the method signature compact and provide access to the block state if needed.
 */
@FunctionalInterface
public interface FluidModel extends BlockModel {
    static FluidModel get(Fluid forFluid) {
        return FluidModelImpl.get(forFluid);
    }

    /**
     * Add a FluidQuadSupplier factory for the given fluid.
     *
     * <p>Accepts a factory so that instances can be recreated when render state
     * is invalidated. This allows implementations to cache sprites or other elements of
     * render state without checking for or handling reloads.
     */
    static void registerFactory(Function<Fluid, FluidModel> factory, ResourceLocation forFluid) {
        FluidModelImpl.registerFactory(factory, forFluid);
    }
}
```

```java
public interface FluidAppearance extends FluidColorProvider, FluidSpriteProvider {
    static FluidAppearance of(FluidColorProvider colorProvider, FluidSpriteProvider spriteProvider) {
        return FluidAppearanceImpl.of(colorProvider, spriteProvider);
    }

    static FluidAppearance get(Fluid fluid) {
        return FluidAppearanceImpl.get(fluid);
    }

    static void register(FluidAppearance appearance, Fluid... fluids) {
        FluidAppearanceImpl.register(appearance, fluids);
    }
```

- FREX includes a FluidModel implementation that emulates vanilla fluid output. (`SimpleFluidModel`)
- Simplifies renderer implementations
- Enables custom fluid models for mods

OTHER STUFF

Item Lights

```java
@FunctionalInterface
public interface HeldItemLightListener {
    /**
     * Use this event to add or modify held item lighting due to effects, worn items
     * or other conditions that aren't completely dependent on held items. The renderer will
     * call this event once per frame after the held light is retrieved for the current player
     * or camera entity.
     *
     * <p>Held lights for non-camera entities are currently not supported.
     *
     * @param holdingEntity The entity that is holding (or not holding) a light provider.
     *                      Currently this is always the camera entity.
     *
     * @param heldStack     The item stack that was used to determine the default result.
     *                      May be empty. If no light was found, will be from the secondary hand.
     *
     * @param defaultResult The light result determined by the renderer. Will be the same in all
     *                      listener invocations. Use this to know if another listener has
     *                      already changed the current result.
     *
     * @param currentResult The light result that should be returned if the listener does not
     *                      modify it. May already have been changed by a prior listener.
     *                      Compare with default result to detect this.
     *
     * @return              The light that should be used. Return {@link ItemLight#NONE} to disable
     *                      held light. Can be modified by subsequent listener invocations.
     */
    ItemLight onGetHeldItemLight(LivingEntity holdingEntity, ItemStack heldStack, ItemLight defaultResult, ItemLight currentResult);

    static void register(HeldItemLightListener listener) {
        HeldItemLightListenerImpl.register(listener);
    }

    /**
     * For use by renderer implementations.
     * Renderers that implement item lights should declare {@link FrexFeature#HELD_ITEM_LIGHTS}.
     */
    static ItemLight apply(LivingEntity holdingEntity, ItemStack heldStack, ItemLight defaultResult) {
        return HeldItemLightListenerImpl.apply(holdingEntity, heldStack, defaultResult);
    }
}
```

- Also accessible via JSON loader
- Implementation is currently optional

```java
  static ItemLight of(float intensity, float red, float green, float blue, boolean worksInFluid, int innerConeAngleDegrees, int outerConeAngleDegrees) {
      innerConeAngleDegrees = Math.min(360, Math.max(1, innerConeAngleDegrees));
      outerConeAngleDegrees = Math.min(360, Math.max(innerConeAngleDegrees, outerConeAngleDegrees));
      return new SimpleItemLight(intensity, red, green, blue, worksInFluid, innerConeAngleDegrees, outerConeAngleDegrees);
  }
```

Dynamic Block Breaking Particles

```
/**
 * Called at the end of block breaking and dust particle initialization to give
 * block models an opportunity to adjust sprite and color based on dynamic state.
 *
 * <p>The particle itself is not exposed to eventually accommodate 3D particles
 * or other novel particle types that may be of a class unknown to the model.
 *
 * @param clientLevel
 * @param blockState
 * @param blockPos
 * @param delegate
 */
default void onNewTerrainParticle(@Nullable ClientLevel clientLevel, BlockState blockState, @Nullable BlockPos blockPos, TerrainParticleDelegate delegate) {
    // NOOP
}

// NB: names are long to reduce risk of conflict with mapped named
interface TerrainParticleDelegate {
    void setModelParticleSprite(TextureAtlasSprite sprite);

    void setModelParticleColor(int color);
}
```

- Part of BlockModel interface
- Lacks hit location of Forge system. This should be added, ideally without breaking abstraction of particles

RenderRegionBakeListener

```
private static final RenderRegionBakeListener listener = new RenderRegionBakeListener() {
    @Override
    public void bake(RenderRegionContext<BlockAndTintGetter> context, BlockStateRenderer blockStateRenderer) {
        final BlockState bs = Blocks.MAGENTA_STAINED_GLASS.defaultBlockState();

        for (int x = 0; x < 16; ++x) {
            for (int z = 0; z < 16; ++z) {
                blockStateRenderer.bake(context.originOffset(x, 0, z), bs);
            }
        }
    }

    @Override
    public BlockAndTintGetter blockViewOverride(BlockAndTintGetter baseView) {
        return new ForwardingBlockView(baseView) {
            @Override
            public BlockState getBlockState(BlockPos blockPos) {
                var result = wrapped.getBlockState(blockPos);

                if (result == null || result.isAir()) {
                    if (blockPos.getY() == 128 & (blockPos.getX() & 0xF0) == 0 & (blockPos.getZ() & 0xF0) == 0) {
                        result = Blocks.MAGENTA_STAINED_GLASS.defaultBlockState();
                    }
                }

                return result;
            }
        };
    }
};

private static final Predicate<? super RenderRegionContext<Level>> predicate = ctx -> {
    return ctx.origin().getY() == 128 & (ctx.origin().getZ() & 0xFF) == 0 & (ctx.origin().getX() & 0xFF) == 0;
};

public static void initialize() {
    RenderRegionBakeListener.register(predicate, listener);
}
```

- Render blocks that don’t exist in world
- Enables large-scale effects without setting block states
- The blocks aren't there - interaction would require extra steps

Varia

- Model Registration - simplified model registration and injection
- RenderLoop Events - compatible alternative to mixing in to LevelRenderer
- SpriteInjector / SpriteFinder - add arbitrary sprites, find sprites from UV
- Pastel Renderer - default implementation, vanilla style, good performance
- Base Implementation Classes
  - Provides most of renderer front-end, used by both Pastel and Canvas
  - Base code model implementation and utilities

Areas of Development

- Physical Textures
  - Add normal/height/roughness/reflectance/roughness/ao/emissive texture maps
  - GLSL specification complete
  - Loader will be format-agnostic
  - Able to derive or transform maps from base textures
  - Handles atlas stitching, UV coordinates shared with vanilla base-color maps
- Better selectivity for material maps, more transformation options
- Expose API fully in entity/particle contexts
  - Enable animation, instancing (somehow)
- APIs for environmental effects (rain, clouds, celestial objects, etc)
- Colored lighting extensions (implementation-dependent)

Q&A