

Toss Master 项目设计文档

❖ 目标与说明

本项目构思时主要考虑高级要求中的两条内容：

- （8分）不依赖现有引擎，采用 iOS/Android 平台实现。
- （7分）与增强现实应用结合。

其余基本要求均已实现。

项目分工：

- 负责碰撞检测，物理运动，主循环设计。
- 负责将代码迁移到 Flutter 框架，处理用户交互和 AR。

💡 整体架构

移动端应用开发框架：Flutter

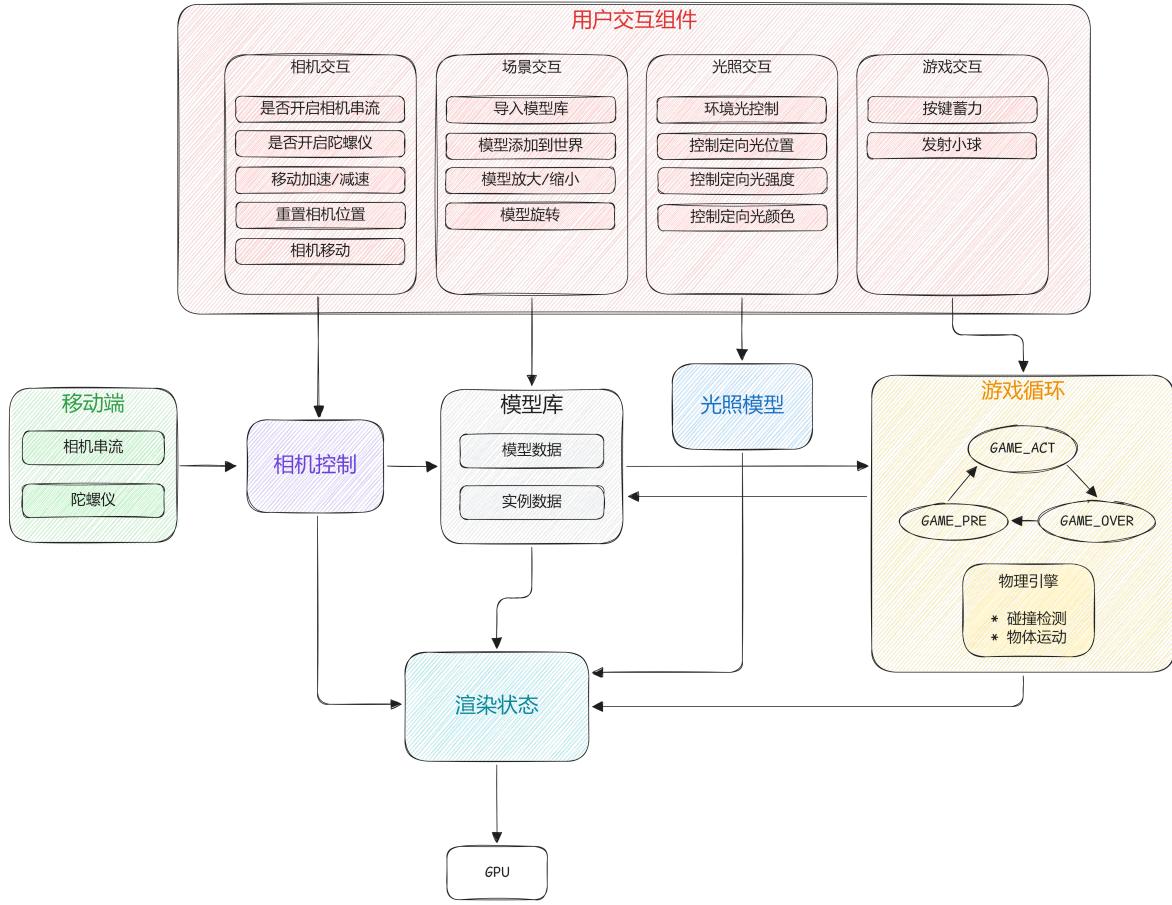
开发语言：Dart

投掷游戏玩法：玩家通过长按蓄力、投掷来命中目标，涉及物理运动，包括重力和碰撞检测。

跨平台实现：利用 Flutter 框架在 Android、HarmonyOS 和 iOS 上完成适配。

增强现实（AR）体验：场景中的物体能够跟随相机视角，营造虚拟与现实共享空间的体验。

整体架构如下图：



整体实现逻辑其实非常清晰：

- **用户交互组件**: 通过用户交互，修改相机参数、模型实例数据、光照模型参数等，进一步地对渲染状态进行修改，更改呈现给用户的画面，从而实现了“交互”功能。
- **移动端调用**: 相机流信号 + 陀螺仪传感器
 - **AR 实现**: 本质是调用移动端的相机信号获取相机串流，并将其渲染为texture，再将 OpenGL 部分与其进行融合。
 - **相机姿态估计**: 由于 `opencv_dart` 缺少关键的相机姿态估计函数 `solvePnP` 和 `estimatePoseSingleMarkers` 的绑定，且OpenCV 相机姿态解析需要先对相机进行大量的标定（Camera Calibration），涉及计算机视觉相关的内容，难以在项目时间内完成。因此最终我们选择自己写一个陀螺仪判定组件，从而实现 AR 最核心的“现实感”（镜头移动，但物体仍保持在原位）
- **相机控制、模型库、游戏循环**这三个部分，作为核心的数据存储/逻辑判定组件，每一个组件都实时对渲染状态进行监控，并各自实现了对渲染状态的异步更新。

OpenGL 部分

模型 with 实例化

项目具备基本的体素建模表达能力，且能够导入 OBJ 格式的三维模型。

- **游戏平台**: 面剔除模式下的立方体表达。
- **游戏投球**: 球体表达。
- **放置在游戏平台上的玩偶**: OBJ模型表达。

与之前的作业实现不同，本次项目中，所有的渲染都基于 `gl.drawArraysInstanced` 函数完成，大大提升了渲染效率，使得项目具备渲染大量物体的能力。为了实现实例化技术，我们也特别设计了模型存储的数据结构。

在项目内，所有模型存储在模型库中。模型库中的每个模型的数据结构具有：

- **模型的基本数据**：顶点、法向量、纹理坐标、纹理图片。
- **模型的实例数据**：对应场景中的每个模型实例，包括位置向量、旋转四元数、缩放数值、特殊标记（标记高亮选中等）。

渲染时，遍历模型库，绘制每个模型的不同实例。

通过实例化渲染，可以减少重复绘制相同模型的开销。

```
for (final model in models) {  
    model.render(gl);  
}  
  
void render(gl) {  
    // 模型数据  
    gl.bindVertexArray(vao);  
    gl.bindBuffer(gl.ARRAY_BUFFER, vbo[0]);  
    gl.bindBuffer(gl.ARRAY_BUFFER, vbo[1]);  
    gl.bindBuffer(gl.ARRAY_BUFFER, vbo[2]);  
    gl.activeTexture(gl.TEXTURE0);  
    gl.bindTexture(gl.TEXTURE_2D, texture);  
    // 实例数据  
    if (instancePosition.isEmpty) return;  
    List<double> mMMatrix = [];  
    for (var i = 0; i < instancePosition.length; i++) {  
        var modelMatrix = Matrix4.compose(instancePosition[i],  
            instanceRotation[i], Vector3.all(instanceScale[i]));  
        mMMatrix.addAll(modelMatrix.storage);  
    }  
    gl.bindBuffer(gl.ARRAY_BUFFER, vbo[3]);  
    gl.bufferData(gl.ARRAY_BUFFER, mMMatrix.length * Float32List.bytesPerElement,  
        Float32List.fromList(mMMatrix), gl.STATIC_DRAW);  
    gl.bindBuffer(gl.ARRAY_BUFFER, vbo[4]);  
    gl.bufferData(  
        gl.ARRAY_BUFFER,  
        instanceFlag.length * Int32List.bytesPerElement,  
        Int32List.fromList(instanceFlag),  
        gl.STATIC_DRAW);  
    // 实例化绘制  
    gl.drawArraysInstanced(  
        gl.TRIANGLES, 0, numVertices, instancePosition.length);  
}
```

几何变换

项目支持用户通过触控手势对模型进行平移、旋转、缩放等基本几何变换。

对于我们渲染的 OpenGL Native Texture 组件，我们将其包装在 Flutter 框架提供的 `GestureDetector` 中，以便捕获用户的触控手势。我们通过该组件的回调函数对模型进行变换：

```

onScaleUpdate: (ScaleUpdateDetails event) => setState(() {
  // 要求已选中特定实例
  if (_selectedModelIndex == null ||
      _selectedModelInstanceIndex == null) {
    return;
  }

  // 移动向量
  var cameraRight = cameraFront.cross(cameraUp);
  cameraRight.normalize();
  var positionDelta =
    cameraRight * (event.focalPointDelta.dx * 0.01) +
    cameraUp * (-event.focalPointDelta.dy * 0.01);

  // 旋转四元数
  var rotationDelta = vm.Quaternion.axisAngle(cameraUp,
    vm.radians((event.rotation - lastRotation) * 50));
  lastRotation = event.rotation;

  // 缩放比例差值
  var scaleDiff = (event.scale - lastScale);
  lastScale = event.scale;

  _models[_selectedModelIndex!].transform(
    _selectedModelInstanceIndex!,
    positionDelta,
    rotationDelta,
    scaleDiff);
}),

```

光照模型

项目实现了 Blinn-Phong 着色的 ADS 光照模型，并实现了基本的光源控制。

本项目有两个光源：

- **全局光**：没有方向，仅有 A（环境光）分量，对每个像素具有相同的光照。
- **定向光（远距离光）**：具有方向和 A（环境光）、D（漫反射光）、S（镜面光）三个反射分量。

光照作为统一变量传递给着色器。

与 Phong 着色相比，Blinn-Phong 着色节省了大量的性能损耗，适合移动端的渲染。

在顶点着色器中，对法向量进行差值，随后在片段着色器中计算 ADS 分量：

```

// normalize the light, normal, and view vectors:
vec3 L = normalize(varyingLightDir);
vec3 N = normalize(varyingNormal);
vec3 V = normalize(-varyingVertPos);

// get the angle between the light and surface normal:
float cosTheta = dot(L,N);

// halfway vector varyingHalfVector was computed in the vertex shader,
// and interpolated prior to reaching the fragment shader.
// It is copied into variable H here for convenience later.

```

```

vec3 H = normalize(varyingHalfVector);

// get angle between the normal and the halfway vector
float cosPhi = dot(H,N);

// compute ADS contributions (per pixel):
vec4 textureColor = texture(s, tc);
vec3 ambient = (globalAmbient.xyz + light.ambient.xyz);
vec3 diffuse = light.diffuse.xyz * max(cosTheta, 0.0);
vec3 specular = light.specular.xyz * pow(max(cosPhi, 0.0), 51.2 * 3.0);

```

用户能够通过「光照」控制界面的控件调节全局光的强度，定向光的位置、颜色、强度等参数。下面以光源位置调节为例，其采用圆形滑块控件，控件的 `onValueChanged` 回调函数中更新光源的位置：

```

SfRadialGauge(
  axes: <RadialAxis>[
    RadialAxis(
      minimum: 0,
      maximum: 360,
      startAngle: 0,
      endAngle: 360,
      showLabels: false,
      showTicks: false,
      pointers: <GaugePointer>[
        MarkerPointer(
          value: (atan2(lightPos.y, lightPos.x) * 180 / pi) %
            360,
          enableDragging: true,
          markerHeight: 20,
          markerWidth: 20,
          markerType: MarkerType.circle,
          color: Colors.red,
          onChanged: (value) {
            setState(() {
              double radians = value * pi / 180;
              lightPos.x = cos(radians) * lightPos.length;
              lightPos.y = sin(radians) * lightPos.length;
            });
          },
        ),
      ],
      annotations: [
        GaugeAnnotation(
          widget: Text("位置"),
          // angle: 90,
        ),
      ],
    ),
  ],
)

```

材质与纹理

项目具备基本的材质和纹理的显示和编辑能力。

在导入模型时，可以选择性地导入纹理图片，存储在上节所述的模型库的每个模型中：

```

factory ImportedModel(
    gl, String objData, Uint8List? texData, Uint8List? gifData) {
// ...
// 纹理数据
int? texture;
if (texData != null) {
    log('ImportedModel.assets() texData');
    Image texImg = decodeImage(texData)!.convert(numChannels: 4);
    texImg = flipVertical(texImg);
    var textureData = NativeUint8Array.from(texImg.toUint8List());
    texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texImg.width, texImg.height, 0,
        gl.RGBA, gl.UNSIGNED_BYTE, textureData);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
}
// ...
}

```

对于所有不具备纹理的模型，我们提供默认的材质。材质按照 ADS 光照模型对材质进行着色：

```

const List<double> goldAmbient = [0.2473, 0.1995, 0.0745, 1.0];
const List<double> goldDiffuse = [0.7516, 0.6065, 0.2265, 1.0];
const List<double> goldSpecular = [0.6283, 0.5559, 0.3661, 1.0];
const double goldShininess = 51.2;

```

材质所使用的着色器：

```

// material
ambient = ((globalAmbient * material.ambient) + (light.ambient *
material.ambient)).xyz;
diffuse = light.diffuse.xyz * material.diffuse.xyz * max(cosTheta, 0.0);
specular = light.specular.xyz * material.specular.xyz * pow(max(cosPhi, 0.0),
material.shininess*3.0);
fragColor = vec4((ambient + diffuse + specular), 1.0);

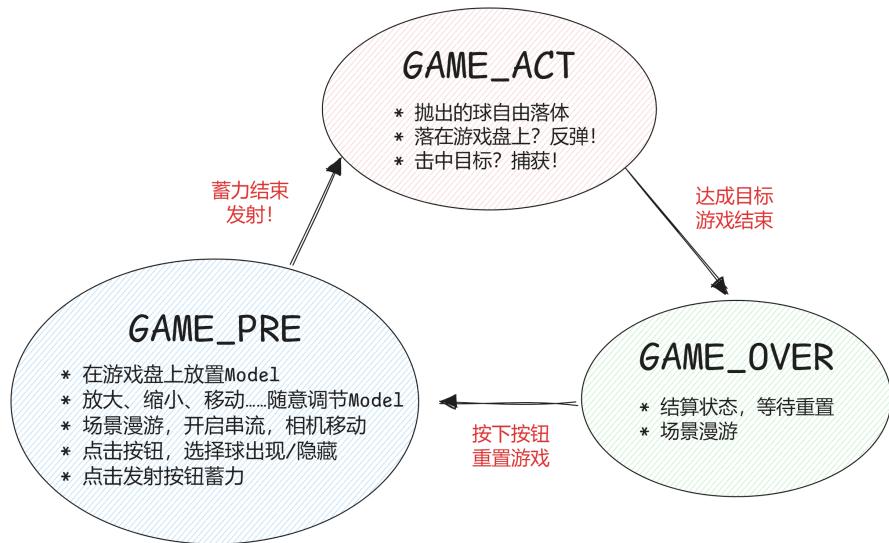
```

我们也测试了结合纹理与光照，但是发现最终效果不好，参数比较难调节，一般情况下都造成模型过暗。所以我们决定分开处理，一个模型要么使用纹理，要么使用材质。

❖ 高级要求部分

漫游时碰撞检测

我们的游戏状态机如下：



由于Dart语言限制，本项目中，我们将碰撞检测算法封装为**模型类的一个成员方法**，并将其作为游戏循环中的一个判定条件，实现游戏状态的转移。

该游戏循环和渲染循环在同一个**异步线程**中执行，因此能够实现**实时的碰撞检测**。

```
GameMode _gameMode = GameMode.pre;
var lastTime = 0;
gameLoop() {
    var currentTime = DateTime.now().millisecondsSinceEpoch;
    var dt = currentTime - lastTime;
    lastTime = currentTime;

    var curr = 0;
    switch (_gameMode) {
        case GameMode.pre:
            // 球体随相机移动
            var rMat = vm.Matrix4.identity();
            rMat.rotate(cameraFront.cross(cameraUp), vm.radians(-35.0));
            var tmp = vm.Vector3.copy(cameraFront);
            tmp.scale(5);
            _sphere.instancePosition[curr] = cameraPos + rMat.transform3(tmp);
            _sphere.instanceScale[curr] = 1.8;
            // 物体运动
            for (var i = 0; i < _models.length; i++) {
                for (var idxModel = 0;
                    idxModel < _models[i].instancePosition.length;
                    idxModel++) {
                    _cuboid.collisionModel(0, _models[i], idxModel);
                }
            }
            break;
        case GameMode.act:
    }
}
```

```

// 球体运动
_sphere.update(dt, gravity);
for (var i = 0; i < _models.length; i++) {
  for (var j = 0; j < _models[i].instancePosition.length; j++) {
    if (_sphere.collisionModel(curr, _models[i], j)) {
      _models[i].instanceFlag[j] = 1;
      _gameMode = GameMode.over;
    } else {
      _models[i].instanceFlag[j] = 0;
    }
  }
}

// 游戏结束：球与板碰撞、球超出视界
_sphere.collisionCuboid(0, _cuboid, 0);
_sphere.collisionCuboid(0, _cuboid, 1);
if (((_sphere.instancePosition[0] - cameraPos).length > 500)) {
  _gameMode = GameMode.over;
}
break;
case GameMode.over:
  break;
}
}

```

碰撞检测算法——AABB & 球

本项目中一共涉及**两类碰撞**：

- 模型与游戏平台的碰撞：AABB & AABB
- 球与模型/游戏平台的碰撞：球 & AABB

算法原理此处不做赘述，其核心函数是 glsl 库的 `clamp` 函数。但在 dart 语言中并没有相关的扩展可以使用，因此我们需要自己实现 `clamp` 函数功能。

整个碰撞算法一共需要完成**两个部分**：

- 计算模型中心 + AABB 的半边长区域/球的半径
- 实现碰撞检测

计算AABB的相关数据，主要是通过遍历模型顶点，分别找到 x 方向、y 方向、z 方向上极大值、极小值，取平均是中心，算距离是边长。

注意并不是所有的OBJ模型中心都在原点，因此在计算完初始模型的中心之后，还应当在模型局部坐标系中，对所有顶点进行一次平移，使得模型在原点。

除此之外，由于我们采用实例化技术，因此实际上对于每个实例，模型库的原始AABB半边长都是相同的，只需要乘上一个 `scale` 数据，就可以得到当前实例对应的半边长大小。

以下是碰撞检测函数（以球碰 AABB 为例）：

```

bool collisionModel(
  int idxBall, // 需要检测碰撞的实例
  ImportedModel model, // 碰撞检测的模型
  int idxBox // 碰撞检测的模型实例
) {
  var diff = instancePosition[idxBall] - model.instancePosition[idxBox];
}

```

```

var clampedX = diff.x;
var clampedY = diff.y;
var clampedZ = diff.z;
var halfSizeX = model.halfSize.x * model.instanceScale[idxBox];
var halfSizeY = model.halfSize.y * model.instanceScale[idxBox];
var halfSizeZ = model.halfSize.z * model.instanceScale[idxBox];

if (clampedX < -halfSizeX) clampedX = -halfSizeX;
if (clampedX > halfSizeX) clampedX = halfSizeX;
if (clampedY < -halfSizeY) clampedY = -halfSizeY;
if (clampedY > halfSizeY) clampedY = halfSizeY;
if (clampedZ < -halfSizeZ) clampedZ = -halfSizeZ;
if (clampedZ > halfSizeZ) clampedZ = halfSizeZ;

diff = Vector3(clampedX, clampedY, clampedZ) - diff;

return diff.length < instanceScale[idxBall];
}

```

移动端跨平台实现

在移动端，我们不再具有 PC 端完善的基础设施，缺少 SOIL2、GLFW、GLM 等库，因此我们需要自行实现这些功能。

首先要考虑的就是**帧缓冲区**。在 PC 上，GLFW 帮助我们完成了窗口和帧缓冲区的创建。但是在移动端，我们需要**自行创建**，`flutter_gl` 仅仅提供将离线渲染好的纹理绘制到屏幕上的功能。因此我们先创建离屏渲染用的 FrameBuffer：

```

setupDefaultFBO() {
    final gl = flutterGlPlugin.gl;
    int glWidth = (width * dpr).toInt();
    int glHeight = (height * dpr).toInt();

    // 离屏渲染用 FBO
    defaultFbo = gl.createFramebuffer();
    gl.bindFramebuffer(gl.FRAMEBUFFER, defaultFbo);
    // 颜色纹理附件
    defaultFboTex = gl.createTexture();
    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, defaultFboTex);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, glWidth, glHeight, 0, gl.RGBA,
        gl.UNSIGNED_BYTE, null);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.framebufferTexture2D(
        gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, defaultFboTex, 0);
    // 缓冲对象附件
    var defaultFboRbo = gl.createRenderbuffer();
    gl.bindRenderbuffer(gl.RENDERBUFFER, defaultFboRbo);
    gl.renderbufferStorage(
        gl.RENDERBUFFER, gl.DEPTH24_STENCIL8, glWidth, glHeight);
    gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_STENCIL_ATTACHMENT,
        gl.RENDERBUFFER, defaultFboRbo);
}

```

完成离屏渲染 FrameBuffer 绑定后，后续的渲染操作都是在这个 FrameBuffer 上进行的，按照 PC 端的渲染流程执行即可。

渲染完成后，调用 `flutter_gl` 提供的函数，将该 FrameBuffer 的颜色纹理传递到原生纹理组件，从而实现渲染到屏幕。

```
gl.finish();
if (!kIsWeb) {
  flutterGlPlugin.updateTexture(defaultFboTex);
}
```

增强现实应用

增强显示主要涉及到**两个关键**：

- 如何将手机拍摄的真实世界与 OpenGL 图形进行融合？
- 如何实现摄像头移动，但模型仍保持在放置的原位？

针对以上两个问题我们采用的**解决方案**是：

- 提取相机流中的图片帧，将每一帧作为 texture 进行渲染，从而实现与 OpenGL 图形的融合。
- 自主实现陀螺仪检测及运算，估算手机摄像头姿态，进而实现“现实感”的模拟。

相机流渲染

首先需要对相机流的传输进行控制，并对每一帧图像进行转码，包括转换图像格式、裁切图像和记录图像数据等操作。

```
void streamCameraImage() {
  if (cameraController?.value.isStreamingImages == false) {
    cameraController?.startImageStream((image) async {
      cameraThrottler.run(() async {
        imglib.Image processedImage = convertCameraImage(image);
        bgWidth = min(processedImage.width, processedImage.height);
        bgHeight = bgWidth;
        // 从中心裁切 bgWidth 的正方形
        int x = (processedImage.width - bgWidth) ~/ 2;
        int y = (processedImage.height - bgWidth) ~/ 2;
        processedImage = imglib.copyCrop(processedImage,
          x: x, y: y, width: bgWidth, height: bgWidth);

        cameraData = NativeUint8Array.from(processedImage.toUint8List());

        // debug: print image and data properties
        developer.log(
          "[startImageStream] image:
${processedImage.width}x${processedImage.height} ${processedImage.format}");
        developer.log("[startImageStream] cameraData: ${cameraData!.length}");
      });
    });
  } else {
    cameraController?.stopImageStream();
    cameraData = null;
  }
}
```

获取了 `cameraData` 之后，将其绑定为纹理，并在渲染循环中不断进行绘制。

```
// 背景纹理渲染
renderBackground() {
    final gl = flutterG1Plugin.gl;
    gl.useProgram(bgProgram);
    // 禁用深度测试
    gl.disable(gl.DEPTH_TEST);
    // 绑定纹理
    gl.bindTexture(gl.TEXTURE_2D, bgTexture);
    // 传递纹理数据
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, bgWidth, bgHeight, 0, gl.RGB,
        gl.UNSIGNED_BYTE, cameraData);
    // 设置纹理单元
    gl.activeTexture(gl.TEXTURE0);
    gl.uniform1i(gl.getUniformLocation(bgProgram, 'bgTexture'), 0);
    // 绘制
    gl.bindVertexArray(bgVao);
    gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);
}
```

陀螺仪信息获取与处理

本项目中，主要通过如下的 `_setupGyroscope` 来实现陀螺仪的事件监听，并通过获取到的陀螺仪的改变来计算出相机旋转矩阵，从而计算出手机导致的相机姿态的变换。

其中 `sensorInterval` 定义了传感器的采样间隔，`_gyroscopeSensitivity` 定义了陀螺仪的灵敏度，并且可以在用户交互界面对陀螺仪灵敏度进行调节。

```
// 陀螺仪
bool _listenGyroscope = false;
GyroscopeEvent? _gyroscopeEvent;
int? _gyroscopeLastInterval;
final _streamSubscriptions = <StreamSubscription<dynamic>>[];
DateTime? _gyroscopeUpdateTime;
static const Duration _ignoreDuration = Duration(milliseconds: 20);
Duration sensorInterval = SensorInterval.gameInterval;
double _gyroscopeSensitivity = 0.5;

_setupGyroscope() {
    _streamSubscriptions.add(
        gyroscopeEventStream(samplingPeriod: sensorInterval).listen(
            (GyroscopeEvent event) {
                final now = event.timestamp;
                setState(() {
                    _gyroscopeEvent = event;
                    if (_gyroscopeUpdateTime != null) {
                        final interval = now.difference(_gyroscopeUpdateTime!);
                        if (interval > _ignoreDuration) {
                            _gyroscopeLastInterval = interval.inMilliseconds;
                        }
                    }
                });
                _gyroscopeUpdateTime = now;

                if (_gyroscopeEvent != null && _listenGyroscope) {
                    final dt = _gyroscopeLastInterval != null

```

```

        ? _gyroscopeLastInterval! / 1000.0
        : 0.033; // 默认 33ms

        // 旋转矩阵
        final rotationMatrix = vm.Matrix4.rotationX(
            _gyroscopeEvent!.x * dt * _gyroscopeSensitivity) *
        vm.Matrix4.rotationY(
            _gyroscopeEvent!.y * dt * _gyroscopeSensitivity) *
        vm.Matrix4.rotationZ(
            _gyroscopeEvent!.z * dt * _gyroscopeSensitivity);

        // 更新相机方向
        cameraFront = rotationMatrix.transform3(cameraFront);
        cameraFront.normalize();
        cameraUp = rotationMatrix.transform3(cameraUp);
        cameraUp.normalize();
    }
},
onError: (e) {
    showDialog(
        context: context,
        builder: (context) {
            return const AlertDialog(
                title: Text("Sensor Not Found"),
                content: Text(
                    "It seems that your device doesn't support Gyroscope Sensor"),
            );
        });
},
cancelOnError: true,
),
);
}

```

★ 项目难点

AR 实现

对于 AR，我们最初计划使用 OpenCV ArUco Marker 标记识别获得相机姿态，但是遇到了下列问题：

- OpenCV VideoCapture 在安卓上仅能输出灰度图像，原因见 [OpenCV](#)。
- `opencv_dart` 缺少关键的相机姿态估计函数 `solvePnP` 和 `estimatePoseSingleMarkers` 的绑定。
- OpenCV 相机姿态解析需要先对相机进行大量的标定（Camera Calibration），这涉及计算机视觉相关的内容，我们无法在短时间内完成。

这些问题非常关键，导致我们放弃了 OpenCV。

接下来考虑现有 AR 框架。AR 框架在平台之间分裂，如安卓有 ARCore，iOS 有 ARKit。引入 AR 框架将导致平台依赖，也加重了负担，因为我们只希望从单张图片中获取相机姿态。如果使用 AR 框架，一般需要启动一个 AR 会话，导致相机的实时捕捉。我们其实也做了一些尝试，但是没有成功：

- `ar_flutter_plugin` 能够支持安卓和 iOS，但已经 2 年无人维护，产生了一堆依赖问题，短时间内我们无法解决。

- `arcore_flutter_plugin` 缺少相机参数接口。
- `arkit_plugin` 具有接口，但开发人员缺少 iOS 设备，无法测试。

因此最终我们选择手动监听陀螺仪的事件，以此计算旋转矩阵，从而修改相机姿态相关数据。

移动端资源限制

PC 端的即时渲染 (IMR, Immediate Mode Rendering)，一次性渲染整个帧缓冲，需要大量的带宽。

这是移动端完全不能支持的资源需求，因此为了满足移动端的渲染流畅，我们采用了分块渲染 (TBR, Tile-Based Rendering) 算法，将帧缓冲分割为一小块一小块，然后逐块进行渲染。

极度缺乏基础库的移动端框架

在跨平台框架如Flutter中，直接使用OpenGL这类底层图形库进行开发并不常见（因为 Flutter 有自带的游戏引擎 Flame），因此缺乏成熟的解决方案和工具链支持。

1. **跨平台兼容性**：Flutter 等框架旨在提供跨平台的UI一致性，但底层图形操作往往需要针对不同平台进行特定的优化和调整。因此我们需要自己解决平台间的差异，如不同的 API 调用和性能特性（甚至包括 iOS 更严苛的代码检查）。
2. **社区和资源**：由于 OpenGL 在 Flutter 中的应用较少，相关的社区支持和资源有限。我们难以找到现成的解决方案，很多地方都需要自己探索和解决问题，这增加了开发的复杂性和难度。
3. **学习曲线**：许多基础的 OpenGL 拓展库在 Flutter 框架中并不成熟，因此许多原本在 OpenGL 中只需要一个函数就能解决的问题，在 Dart 语言中可能需要数倍的代码量才能完成（当然这也受限于 Dart 语言的奇妙特性），这导致我们不得不去挖掘许多底层的设计算法，并且研究 Dart 语言本身就已经加长了整个学习曲线。

百花齐放的图像编码 & YUV420 糟糕的访存模式

从 `startImageStream((image) async {})` 获得的 `image` 可能为：

- iOS: BGRA8888
- Android: YUV420 (适用于视频流的一种编码，将明度与颜色分开存储，在低带宽时能够只显示黑白画面)

然而 OpenGL `glTexImage2D` 只支持 RGB、RGBA 等格式。

这导致我们不仅需要自主实现图像转码，还需要区分不同系统的编码模式。

抛开这一点不说，就安卓的 YUV420 而言，它将亮度信息和色度信息分开存储，以减少所需的带宽。然而，这种分离存储也意味着在处理图像时需要更多的计算来重新组合这些信息，让本就不高的带宽雪上加霜。

```
for (int h = 0; h < imageHeight; h++) {
    int uvh = (h / 2).floor();
    for (int w = 0; w < imageWidth; w++) {
        int uvw = (w / 2).floor();
        final yIndex = (h * yRowStride) + (w * yPixelStride);
        final int y = yBuffer[yIndex];
        final int uvIndex = (uvh * uvRowStride) + (uvw * uvPixelStride);
        final int u = uBuffer[uvIndex]; final int v = vBuffer[uvIndex];
        int r = (y + v * 1436 / 1024 - 179).round();
```

```
int g = (y - u * 46549 / 131072 + 44 - v * 93604 / 131072 + 91).round();
int b = (y + u * 1814 / 1024 - 227).round();
r = r.clamp(0, 255); g = g.clamp(0, 255); b = b.clamp(0, 255);
image.setPixelRgb(imageHeight - h - 1, imageWidth - w - 1, r, g, b);
}
}
```

参考文献

- 下面的文献帮助我们学习了如何使用 Flutter Camera 库的相机提供图片流，并进行适当的编码处理供 OpenGL 渲染。
 - [theamorn/flutter-stream-image](#)
 - [\[Flutter\] 用相機畫面一小部分做辨識。這篇文章源自於我在工作上第一次選用 Flutter... | by Claire Liu | Flutter Taipei | Medium](#)
 - [image_converter.dart](#)
- OpenGL 部分主要参考教程：[主页 - LearnOpenGL CN](#)