1. **Arcane Gambit Computer Vision Module**

Overview:

Arcane Gambit is a DND-like PvP card game. This module contains the necessary functions and methods to communicate the real-world game session with Unreal Engine to enforce game logic and rules. The module leverages computer vision techniques to detect and classify game elements such as dice and cards, ensuring accurate gameplay.

Header:

```
#pragma once

#ifdef ARCANE_DLL_EXPORTS
#define ARCANE_DLL_API __declspec(dllexport)
#else
#define ARCANE_DLL_API __declspec(dllimport)
#endif

#include <opencv2/core.hpp>

// ----------------------------
// Detection & Classification
// ----------------------------
#define DICE_CLASS 0

struct DetectionResult {
cv::Rect boundingBox;
int classId;
float confidence;
};

struct DetectionResultArray {
DetectionResult* results;
int size;
};

struct ClassificationResult {
int classId;
float confidence;
};

extern "C" {
ARCANE_DLL_API bool InitializeNetworks(const char* detectionModelPath, const char* classificationModelPath, bool
useCuda);
ARCANE_DLL_API DetectionResultArray Detect(const cv::Mat& frame);
ARCANE_DLL_API ClassificationResult Classify(const cv::Mat& frame);
ARCANE_DLL_API void Cleanup();
}
```

```
// ----------------------------
// PlayerReady Integration
// ----------------------------
extern "C" {
// Create and destroy PlayerReady instance
ARCANE_DLL_API void* CreatePlayerReady(int roiRadius, cv::Point center);
ARCANE_DLL_API void DestroyPlayerReady(void* instance);

// Set frame to PlayerReady
ARCANE_DLL_API void PlayerReady_SetFrame(void* instance, const cv::Mat& frame);

// Check if the green area is covered long enough
ARCANE_DLL_API bool PlayerReady_CheckReady(void* instance, int secondsToCheck);
}
```

```
// Function prototypes
bool checkCudaComputeCapability();
int classConverter(int classId);
void classifyImage(Net& netClassification, const Mat& image);
Net initializeNetwork(const string& modelPath, bool useCuda);
Mat preprocessImage(const Mat& frame, const Size& targetSize, bool useCuda);
Mat scaleToFitScreen(const Mat& image);
Rect adjustToSquare(const Rect& box, int frameWidth, int frameHeight);
vector<Rect> processDetections(const Mat& output, const Mat& frame, vector<float>& confidences);
DetectionResult classifyRegion(const Mat& cropped, const Rect& box, Net& netClassification, int
frameWidth, int frameHeight);
vector<DetectionResult> runDetection(Net& netDetection, Net& netClassification, const Mat& frame, bool
useCuda);
vector<DetectionResult> detectDice(Net& netDetection, Net& netClassification, const Mat& frame, bool
useCuda);
Mat generateFrame(const Mat& image, const vector<Rect>& boxes, const vector<float>& confidences);
```

Features:

1. **Real-Time Detection:**
   - Detects dice in live video streams using YOLO-based object detection.
   - Supports GPU acceleration with CUDA for faster processing.
2. **Image Classification:**
   - Classifies dice or cards based on their visual features.
   - Uses ONNX models for inference.
3. **Modularity:**
   - Detection and Classification models are separated modularly, they can be retrained without harming the other.

4. **Cross-Platform Compatibility:**
    - Developed as a DLL for seamless integration with Unreal Engine or other C++ applications.
    - Designed to work on both CUDA-enabled and non-CUDA systems, with CUDA-enabled systems achieving up to 20x performance improvement.

What We've Done:

1. Trained models for detection and classification for D20 Dice
2. Converted torchscript (pt) models to more compatible ONNX models and tested their accuracy
3. Implemented driver code on C++ for inference.
4. Integrated CUDA for GPU acceleration.
5. Designed a modular API for easy integration with external systems.
6. Tested the module with sample images and live video streams using DLL.

Problems Faced:

1. **CUDA Compatibility**:
    - Ensuring the module works with different CUDA versions and devices.
    - Solution: Added a checkCudaComputeCapability function to verify device compatibility (Currently Compute Capability 8.6 – RTX 3050 and Newer).
2. **OpenCV Compatibility:**
    - OpenCV does not provide CUDA support, compiling from source is mandatory. Compiled OpenCV binaries (4.11.0) had issues with YOLO11 models.
    - Solution: Compiled OpenCV from source, retrained the models using YOLOV8
3. **Model Loading Issues**:
    - ONNX models failed to load in some cases due to incorrect paths or unsupported layers.
    - Solution: Verified model paths and ensured compatibility with OpenCV's DNN module.
4. **Real-Time Performance**:
    - Achieving low latency in live detection mode.
    - Solution: Optimized preprocessing and enabled CUDA acceleration, Switched to multithreaded process model.
5. **Memory Management**:
    - Ensuring proper cleanup of resources to avoid memory leaks.

    - Solution: Added a Cleanup function to release networks and reset the CUDA device.
6. **Integration Challenges**:
    - Ensuring seamless integration with Unreal Engine.
    - Solution: Built the module as a DLL with a clear API.

PlayerReady Class

Overview:

Checks whether a green-colored area covers a circular region of interest (ROI) in the image for a specified duration. Useful for determining if a player is ready based on visual cues.

```
#ifndef PLAYER_READY_H
#define PLAYER_READY_H

#include <opencv2/opencv.hpp>
#include <chrono>

class PlayerReady {
public:
```

```
PlayerReady(int roiRadius, cv::Point center);

void setFrame(const cv::Mat& newFrame);
bool checkReady(int secondsToCheck = 3);

private:
cv::Mat frame;
cv::Point center;
int roiRadius;

cv::Scalar lowerGreen = cv::Scalar(40, 50, 50);
cv::Scalar upperGreen = cv::Scalar(80, 255, 255);

bool isGreenCovered();
};

#endif // PLAYER_READY_H
```

1. Constructor:

```
PlayerReady(int roiRadius, cv::Point center);
```

a. This constructor takes two parameters:
   i. roiRadius: The radius of the region of interest (ROI) that will be checked for green coverage.
   ii. center: A cv::Point representing the center of the ROI. This value is passed as an argument when an instance of PlayerReady is created.

b. The constructor initializes these values and sets the stage for further processing.

2. setFrame method:

```
void setFrame(const cv::Mat& newFrame);
```

a. This method is used to set the frame to be processed.
b. The frame object is updated with the content of newFrame, which is a cv::Mat object representing the current image or video frame.
c. The setFrame method does not modify the center or roiRadius, as these are set via the constructor.

3. checkReady method:

```
bool checkReady(int secondsToCheck = 3);
```

a. This method checks if the green area has been covered for a specified number of seconds.
b. The default value for secondsToCheck is 3 seconds.
c. It uses a loop to continuously check the frame and the ROI until the specified time has passed (if the green area is sufficiently covered).
d. If the green area is covered for the required time, it returns true, indicating the player is ready.
e. If the green area is not covered, the timer is reset.

4. Private Members:

```
cv::Mat frame;
cv::Point center;
int roiRadius;

cv::Scalar lowerGreen = cv::Scalar(40, 50, 50);
cv::Scalar upperGreen = cv::Scalar(80, 255, 255);

bool isGreenCovered();
};
```

a. **frame**: A cv::Mat object that stores the current frame to be processed.
b. **center**: A cv::Point object that defines the center of the ROI. This is used to create the circular mask for detecting the green area.
c. **roiRadius**: An integer that defines the radius of the ROI for green detection.
d. **lowerGreen and upperGreen**: These cv::Scalar values define the HSV color range for detecting green pixels in the image.
e. **isGreenCovered()**: A helper method that checks if the ROI contains enough green pixels, determining whether the green area is sufficiently covered in the region.