Boston University

Electrical & Computer Engineering

EC463 Senior Design Project

First Semester Report

**Mario Kart 64 Analytics Engine**

**Submitted to**

Dan Ryan

Bytelight

183 State Street

7th Floor

(617) 828-6218

dpryan87@gmail.com



**by**

Team 15

The Karty Boyz

**Team Members:**

Michael Abed [mgabed@bu.edu](mailto:mgabed@bu.edu)

Jonathan Bell [jbell@bu.edu](mailto:jbell@bu.edu)

Johan Mickos [jmickos@bu.edu](mailto:jmickos@bu.edu)

Josh Navon [navonj@bu.edu](mailto:navonj@bu.edu)

Mark O’Brien [mark0b@bu.edu](mailto:mark0b@bu.edu)

For the Period of Performance:

December 10, 2013 – May 1, 2014

Submitted: December 10, 2013

**Table of Contents**

[Executive Summary](#h.a4t7omwmsj5n)

[Introduction](#h.l84n5k6h3gin)

[Concept Development](#h.dyry9w728iu7)

[Conceptual Approach](#h.hy92uh43a92a)

[Alternative Solutions](#h.ppkz77x2rm4v)

[System Description](#h.e5doq1v30n78)

[User Interface](#h.lynvyxe5x3an)

[Behavior](#h.qx65lul5fevu)

[Technology Selection](#h.cjyg6wls8hyz)

[Storage API](#h.s40pxeq2s1ct)

[Behavior](#h.q74hyoiiwbun)

[Technology Selection](#h.qkp6691d66jo)

[Data Processing](#h.88s0ehzjgia)

[Behavior](#h.82o6ofuq8ad4)

[Technology Selection](#h.pqm2a1jpj4uj)

[Core Server](#h.hw9ayztlg5y6)

[Behavior](#h.g6puguy0l5k5)

[Technology Selection](#h.gilu1twtj51b)

[First Semester Progress](#h.yqfedb4j1ga9)

[Technical Plan](#h.ns2nb4yld9p)

[Video processing](#h.fd16ah8v0s7b)

[Storage API](#h.yo9er8ex6eqc)

[Core Server](#h.kqqgv2ajbvan)

[Budget Estimate](#h.cljcbexzhz5s)

[Already spent](#h.493g15vi1x9d)

[Planned Future Spending](#h.jl4zk8x6hub5)

[Possible Future Spending](#h.5lkkga459r2l)

[Attachments](#h.iljje6jf6p9s)

[Appendix 1 — Engineering Requirements](#h.kukvywmijvw)

[Appendix 2 — Gantt Chart](#h.ckqttnxsrlci)

# Executive Summary

Mario Kart 64 Analytics Engine  
Team 15 — The Karty Boyz

The customer, an avid group of Mario Karters, has no way to review the events of old races. They wish to have a system from which they can query their statistics, or watch previous races. The proposed solution is two fold: a processing suite implemented through OpenCV, and a set of servers that will upload the video, launch computation on the video, and store both the events and video in a predefined database. These servers will be implemented through the hardware and API of the Massachusetts Open Cloud. This innovative system will dynamically scale to the user’s demand, and spin up extra nodes for computation if the processing node is flooded with requests. Our processing program will use a set of individually designed masks, created specifically for this project.

# Introduction

Dan Ryan, the customer, is an avid Mario Kart aficionado. A few days a week he plays Mario Kart with friends. He wishes to have a way to track his statistics from played races, both in order to understand the underlying mechanics better, and to keep a leaderboard for him and his friends. Mr. Ryan has tasked us with creating a queriable database that stores race events. An event in Mario Kart is an action from a predefined list of possible actions. Every action has a specific associated sprite that is displayed each time the action is performed. An example of an action is a character picking up an item. The sprite associated with this action is the item box appearing at the top of the specific character’s window.

Detection of all specified events is the first, and most important task for this project. Event detection will be implemented through the program OpenCV. We will use mask detection to recognize events. The mask for the example event above would be a pixelated version of the item box. A parent class has been created that has a handler function per implemented mask. Processing is done on each frame sequentially; however not every handler is called on every frame. The parent class follows a set of predefined rules in order to call the masks for events possible in the current frame. Once an event has been detected, a JSON describing the event is POSTed to the database.

Once the event JSON is POSTed, it interacts with the second aspect of our project, the storage. We have two divisions of our storage, information storage and video storage. We are creating a Cinder block volume to store the information. There are three types of stored information: event information (which puts data in the session, race, and event table), postgresql database architecture descriptions, and pointers to the uploaded video files. The video files will be stored in a separate blob storage, instantiated as a swift volume, and can be accessed through the pointers in our main database. Once instantiated inside the database, both the event data and the videos will be queriable through the core server’s API.

Our core server will do three things: upload a video, pass the video pointer to the compute node and start a processing job, and query the database for event information. The first two processes are relatively simple and already implemented. When the server receives an upload through the form on it’s website, it POSTs the video to the database, which creates a new entry in the session table, and POSTs the information back to the core server. The core server then passes the pointer to the video file to the compute node, which launches the process to find the start and end of each race. The compute node sends this information to the database, and launches a separate process for each race. The core server’s website will be able to query the database through two interfaces: a simple box interface for basic queries, and a more advanced text field in order to provide elasticity, and allow more complex queries.

Through our implementation, the users will be able to query their statistics on their beloved game: Mario Kart 64. We provide a valuable service to those who have spent many hours with a trident controller held in their hand: validation for their hours perfecting their drift technique. They can finally lord their superior scores over their friends. Finally, cries of “You always get the &@#$ing lightning!” can be met with “Oh yeah!?! Prove it!”

# Concept Development

The customer's problem stems from a desire to better understand the underlying mechanics of Mario Kart 64. The final product must address the issues of acquiring the video of Mario Kart 64 sessions, analyze the video to identify various game events, and allow the user to examine these events for unique correlations. From an engineering perspective, this boils down to three distinct fields of study:

* Video processing
* Feature detection
* Information retrieval

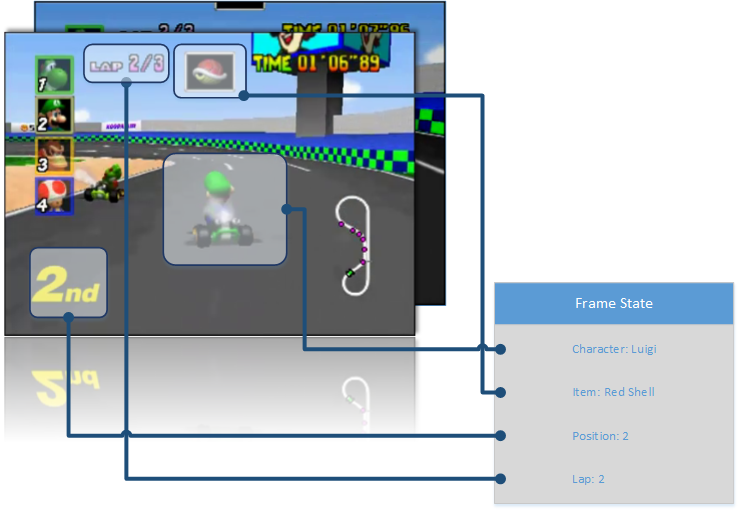
The problem of video processing arises from the need to analyze and interpret the video stream from the game. This is tied to the problem of feature detection: key sprites[[1]](#footnote-0) in the game correspond to specific events and must be identified as such. Information retrieval is a paramount component of the problem, as the final product will need to be able to generate a context subset of the universe of game events. The information retrieval system will be queried to display specific information regarding (possibly unrelated) game events. The requirements regarding these issues can be found in Appendix 1.

## Conceptual Approach

The proposed solution is a user-driven product capable of analyzing uploaded Mario Kart 64 videos and contextualizing event queries into intelligible results. The video processing treats the uploaded video as a series of races. These races consist of event data: state variables for every frame describing the events currently present. This data is stored in a database as collections of events in the respective races. The user can then probe these events in a unique query language to explore aspects of the video game not previously connected, such as the probability of acquiring a certain item when in a specified position in the race.

Each event object contains specific information about when and where it occurs, as well as data concerning *what* type of event it is. The client expressed great interest in the ability to manually mark events within a race by saying a key phrase (“Tag it!” or “Watch this!”). This introduces the concept of audio awareness in addition to the event awareness. The processing suite will also contain a separate module for sound processing.

This particular solution was chosen because of its modularity. All data structures implemented can be represented as objects, allowing for a scalable design. This means that the solution can efficiently be expanded to support other video games that implement billboarding[[2]](#footnote-1). Event detection, for example, relies on two objects: a processing Engine and a collection of Detectors. The engine would iterate through the video frames, applying the Detectors to each frame. Each Detector object contains functions for detecting a given event and for handling the event. By using this approach, the developer can easily add more Detectors to the Engine object without the need for writing exhaustive amounts of code.



*Figure 1: Example of Frame State Variables*

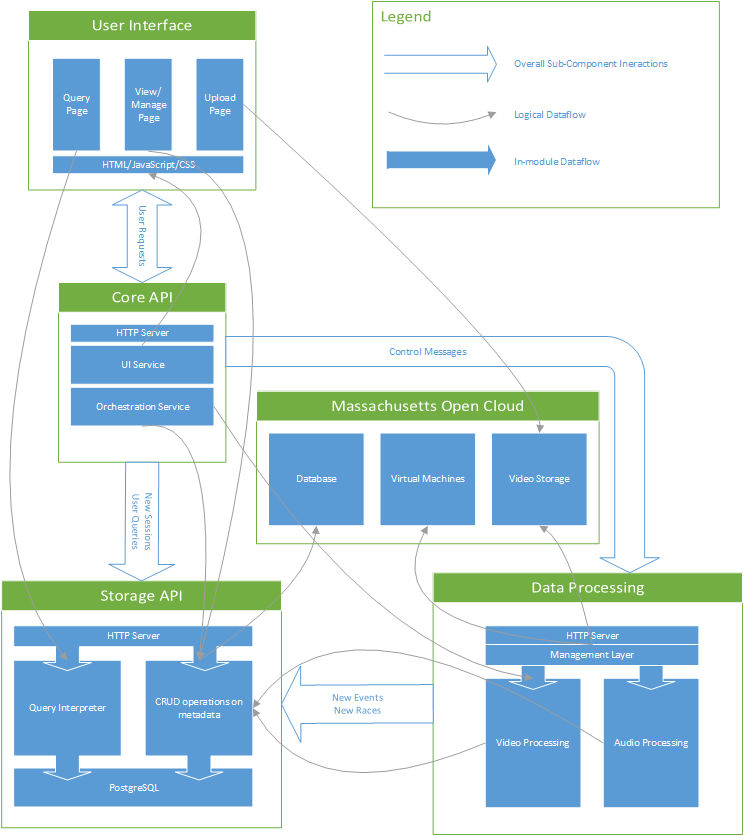
## Alternative Solutions

This object-oriented approach is naturally derived from the structure of video games (where objects are interacting with each other). Consequently, few alternatives to the video processing aspect were considered. At the start of the project, however, we considered alternative approaches to grabbing the video frames from the Nintendo 64. One approach, for example, was creating a device which would extract the composite video signal from the console and convert this to a computer-readable USB format. This idea was quickly discarded due to the excessive time and complexity it required. Furthermore, these devices already exist in the market.

We also toyed with the idea of physically hacking the console in order to extract information directly from the game cartridge and console memory. This idea, however, deviated slightly too far from the goal of the project, and was therefore also discarded. The final solution we are committed to relies on the user capturing the Mario Kart 64 video herself and uploading it to the web server, from which the processing will begin.

# System Description

In order to separate concerns there are 4 primary systems in our design. These are a user interface, a storage system and API, a data processing component, and a core component that facilitates the interactions of the previous 3. The block diagram below shows the overall structure and logical interactions between these various components.



*Figure 2: Systems Diagram*

## User Interface

### Behavior

The user-interface will be where a user interacts with the system both to input their data, and to view the results of the processing on their data.

### Technology Selection

Several possibilities were considered for the interface, but a web-based solution was decided upon based on its ease of user accessibility (i.e. it can be used by any computer with a web browser and internet connection), and its relatively quick development times.

## Storage API

### Behavior

The storage API will manage persistence of metadata detected about user-supplied race videos. This metadata includes the particular video uploaded, referred to as a session, the races contained in this session video, and the events detected in each race. This API will support CRUD[[3]](#footnote-2) operations to manage the metadata.

In addition, the storage API will manage interpreting and responding to user queries. User queries are planned to be relatively free-form, both in the types of questions that can be asked and the nature of the response that is given. In order to accommodate this wide range of inputs, interpreting user queries will be split into two components: expectation detection and subject identification. In expectation detection the actual question that is being asked will be deduced. This will determine if the answer should take the form of a statistic, a specific event, or a specific race, and will decide what operations to do on the result set to get to this result. In subject identification the necessary result set will be determined and the query constraints will be designed. In practical terms, this means that expectation detection will generate the columns desired for a SQL SELECT statement, and the subject identification will generate the WHERE clause.

### Technology Selection

For metadata persistence, a SQL database was determined to be the best option considering the volume of data to be stored, the relational nature of the data, and the unknown nature of most queries. PostgreSQL was selected because of its strict adherence to SQL semantics compared to other popular DBMSs[[4]](#footnote-3) (e.g. MySQL).

Other components of the storage API are less critical and were chosen largely on convenience. A Python library called Flask is being used to create the HTTP APIs, and SQLAlchemy is being used as an object relational mapper, marshalling Python objects to and from PostgreSQL.

## Data Processing

### Behavior

Data processing operates on use-uploaded videos to detect interesting features of them. Three phases were devised to simplify the problem of performing this processing.

In the first phase, boundaries between races are detected and upon the detection of the last frame of a race, metadata for the race is added to the database. This allows the user’s video to be split into videos of its constituent races. When this phase completes, there will be several videos that can be processed in parallel for the events that they contain.

Each race is then processed frame-by-frame. Each frame has several detectors applied to it, each of which is responsible for determining if the frame it is applied on contains the event it is responsible for. For example, an item detector will detect on each frame if a new item has been acquired. If a detector has a positive match, it will create an event in the database.

In addition to detecting events in the video stream, other events in the audio stream will be detected. This will be done by looking at the user-supplied microphone track, and using a trained hidden markov model to detect the phrases “tag it” and “watch this.” When these phrases are detected, they are added to the database.

### Technology Selection

Video processing is being done using OpenCV and Python. This was the natural choice given the quality of OpenCV and the familiarity with Python brought from other components of the project.

The audio processing is being implemented on top of a python library for hidden markov models.

## Core Server

### Behavior

The core server’s job is to serve as a bridge between the user-interface and the storage and data processing subsystems. It orchestrates the behavior of these subsystems, interacting with metadata on the user’s behalf and informing the data processing component of what data it should work on.

The core server can be seen as the entry point to the analytics engine. As a result, it has the dual responsibility of presenting the user-interface and shielding the rest of the system from potentially malicious (or at least malformed) inputs.

On the orchestration side, it will have to take a valid user request, and translate it into either a query, a metadata operation, or the uploading of a new video. In the case of the first two, after ensuring the validity and permission of the user making the request, the core server will forward it effectively unchanged to the storage API. In the case of a new video being uploaded, the core server will verify that the video has been saved and instruct the data processing component to process the new video.

### Technology Selection

The core server is being developed in PHP because of its tight integration with HTML. This will allow many core services to be developed in line with the user-interface. Because not all requests will need to be handled by the core server itself (e.g. requests for video files), it will run behind a web server which will route requests accordingly. Nginx[[5]](#footnote-4) was selected as this web server because of its extensibility and the existence of plugins allowing for easy file uploading and simple streaming of media.

# 

# First Semester Progress

Within the first semester of Senior Design, progress has been made corresponding with all general design tasks. These tasks include event detection, video storage, event information storage, and user interface design. Although the progress for these areas has been generally presented in the form of a first deliverable test, the following will provide a more detailed description of the progress that has been made.

With regards to event detection, the first deliverable test has shown that race start and end events have been detected with a 100% success rate, and accuracy of these event times are within the allowed deviation. The development that has allowed this detection to occur involves the comparison of regions of interest in a game frame that is being tested, with a predetermined mask of the cue that occurs in the video corresponding to the event being detected. These detections convey that the parent class for event detection is functional. This will be used in the detection of all events.

Beyond these functioning detections, development has been made to implement the specific handlers for other event types. These handlers account for differences in the behavior of event cues. For example debouncing must be done on the detection of items because when a player receives an item, there is a sequence of cues that allude to an item being received. These cues include a fast scroll through all available item sprites to visually represent the randomness of getting a specific item. Once this scroll is done, the selected item flashes a few times. The variation of cuing sequences between events necessitates these specific handlers for each event type.

Separate from video based event detection, the functional approach for audio based event detection has been determined. This approach recognizes start and endpoints of a word and splits the full signal of external audio to a set of utterances. The utterances are then converted to observation sequences using cepstral analysis. Once observation sequences are made, desired events are detected using a hidden markov model whose parameters have been trained with a set of observation sequences which are known to contain the cue.

With regards to video storage, the first deliverable test has shown that a process for video storage has been developed using a web server. This process emulates the actual video storage that will be done on the Massachusetts Open Cloud by using a web server and local storage to save and serve video content. Given the difference between these two approaches it is expected that minimal work will need to be done when this work is all shifted to the MOC. Virtual machines and a user account have already been instantiated for the project, but as the MOC machines will not be operable until next semester, the work is currently being done locally.

Information storage is possibly the most imperative portion of the project. During the first deliverable test, once a video is uploaded, it’s corresponding information about the race session is stored in the PostgreSQL database, demonstrating it’s functionality. Once the session has been split, new sets of information for each race that has been detected are also made. These sets of information are then stored into the database for future reference.

This functionality of information storage is made possible by the development that has been done of the storage API. This API is a REST interface which handles requests to create, modify, view and delete database entries. At it’s current state, the API works very well and provides a strong foundation for future development.

With respect to user interface design, the first deliverable testing shows that an interface for video upload has been created. On top of this, testing also conveys how a minimal interface for displaying database entries has been implemented. This will be the foundation of the more aesthetically pleasing interface for viewing data to be developed in the future.

The functionality of the user interface relies greatly on the development of other project tasks. For this reason the approach being followed to implement the user interface, is adding functionalities based on the completion of other tasks. Proactive development of the user interface will be more aggressively pursued next semester.

# 

# 

# Technical Plan

## Video processing

Task 1. Unit test suite

A robust unit test suite will be developed to test the various event detectors. The test suite will run various masks against our event detectors, checking for correct outputs.This should encompass all event detectors already created, and any that are made hereafter. Upon detection, respective state variables will be updated. Leads: Johan Mickos & Josh Navon.

Task 2. Number of players detector

An event detector that checks for the number of players in a match will be designed. It will look for defining characteristics of a four player match compared to a three player match. If time allows, we will also add support for one and two player detection. Upon detection, respective state variables will be updated. Lead: Johan Mickos.

Task 3. Lap change detector

An event detector that checks for a change in the current lap number for each player. The information produced from this detector will aid in lap and race time calculation. This detector will look for the change in number on the lap indicator. Alternatively, Lakitu can be an indicator of a lap change. Upon detection, respective state variables will be updated. Lead: Josh Navon.

Task 4. Item acquisition & use detector

This event detector will check what item the user has received, and also check for when the item is used. Debouncing will have to occur in order to mitigate the roulette effect of receiving an item box. The item use portion will look for the moment that the item box disappears in order to determine when the item has been used. Upon detection, respective state variables will be updated. Leads: Johan Mickos & Josh Navon.

Task 5. Position change detector

An event detector for finding changes in place throughout a race. Each time a user changes place, the event will be triggered. The event will be triggered on a player by player basis. This detector will monitor the status of the place indicator in each players’ screen. Upon detection, respective state variables will be updated. Leads: Johan Mickos & Josh Navon.

Task 6. Collision detection

An event detector that checks for when a player is hit by a shell, banana, bomb-omb (VS Mode only), or fake item box. Unique cues in each event will be singled out and used to identify them. Upon detection, respective state variables will be updated. Leads: Johan Mickos & Josh Navon.

Task 7. Boost detection

This event detector will check for when a player executes a drift boost. This event could be triggered using audio cues from each player, or video cues with unique sprites. Upon detection, respective state variables will be updated. Leads: Johan Mickos & Josh Navon.

Task 8. Shortcut detection

An event detector to check for if a player makes the shortcut on Koopa Troopa Beach. Useful masks could be the view of the cave or a change in place in the map. Upon detection, respective state variables will be updated. Leads: Johan Mickos & Josh Navon.

## Storage API

Task 9. Event API

A REST API for event metadata must be created in a similar vein to how the session and race APIs currently exist. Other subcomponents should be able to create and manage events through performing standard HTTP verbs on various URL endpoints. Leads: Michael Abed

Task 10. Completed Database Schema

The database schema needs to be finalized. This includes a table for users, and a propagation of ownership information to sessions. Additional columns for players, characters, map, and race results must be added. The APIs must be updated to allow access to these fields. Leads: Michael Abed

Task 11. Query Language

More complicated queries will be performed through a more complicated query language. This query language still must be designed and implemented, but its overall functionality will be to allow a user to ask very specific questions about Mario Kart events and get detailed answers. Leads: Michael Abed

## Core Server

Task 12. Box query interface

For simple queries, it is desired that the user be able to perform their query through a simple HTML form based interface. An HTTP webpage will be created to display the querying interface. The query selected by the form will be translated into the designed query language and performed on the storage API. The result of this query will then be nicely displayed for the user. Leads: Jonathan Bell

Task 13. Plaintext query interface

More complicated queries will use the designed query language directly. A page will be created that allows users to enter query strings. After validation and normalization has been performed these queries will be sent to the storage API for execution and the results will be nicely rendered for the user. Leads: Michael Abed, Jonathan Bell

Task 14. User friendly start page

An easy to use web-page will be created, through which the user can sign in, query the database for his or all information, upload a video, or watch one of his previous races. Leads: Jonathan Bell

Task 15. Video upload and play

An additional webpage will be created, which will allow the user to upload a video or play a selected race from the database. Once all other tasks are completed, an additional feature of the video player will be for user’s to select “Exciting Mode”, where the player will automatically cycle through exciting moments. This will be initially chosen through videos marked exciting through “Tag it” or “Watch this”, but the exciting algorithm may be improved. Leads: Jonathan Bell

# 

# 

# Budget Estimate

## Already spent

Sabrent Video Capture: $27.27

Nintendo 64 + controllers + Mario Kart 64: $126.60

**Total: $153.87**

## Planned Future Spending

Microphone Component: $20

Microphone supporting components: $20

**Total: $40.00**

## Possible Future Spending

Amazon AWS (in the event of MOC not working out): $100.00

**Total: $100.00**

# 

# 

# Attachments

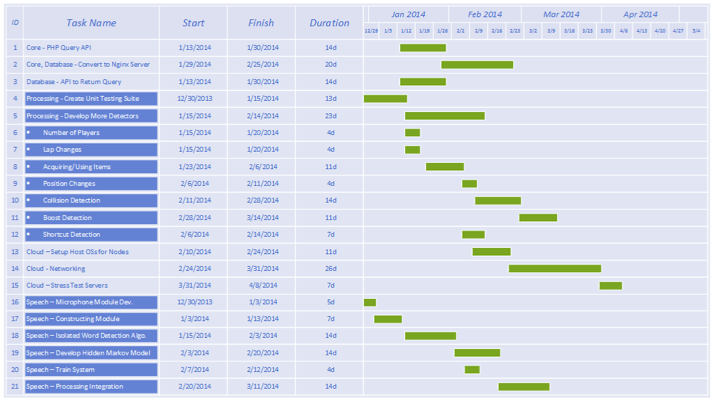
## Appendix 1 — Engineering Requirements

* Events must be detected with:
  + False positive rate of 5% for non-core detectors.
  + Detection rate of 90% for non-core detectors.
  + False positive rate of 0% for start and finish detectors
* Must Recognize These Types of Events:
  + Changing places
  + Game begin/end
  + Lap changes
  + Collisions
  + Getting Item
  + Shortcut detection (course specific)
  + Boost detection (starting boost, mushroom, and drifting)
* Exciting Moment Tagging
* Speech recognition for “Tag it!” and “Watch this!”
  + False negative rate < 3% & False positive rate < 15%.
  + Tag durations where volume is over some (TBD) threshold
* Database (DB)
  + Race events are stored in a table
  + Statistics can be queried in an easy (non-SQL) manner
* Cloud Server
  + Processes a race in less than 2x the race duration
  + Can separate a video of many races into individual race videos
  + Users can upload race videos for processing
* Web interface
  + User can access videos
  + User can query database

## 

## 

## Appendix 2 — Gantt Chart



1. Sprite: a computer graphic that may be moved on-screen and otherwise manipulated as a single entity. [↑](#footnote-ref-0)
2. Billboarding: A technique in three-dimensiona[l](http://en.wiktionary.org/wiki/three-dimensional) graphic[s](http://en.wiktionary.org/wiki/graphic) in which a sprite is rendered perpendicula[r](http://en.wiktionary.org/wiki/perpendicular) to the camer[a](http://en.wiktionary.org/wiki/camera) without respect to camera movement. [↑](#footnote-ref-1)
3. Create, Read, Update, Destroy [↑](#footnote-ref-2)
4. Database Management System [↑](#footnote-ref-3)
5. Pronounced engine-ex [↑](#footnote-ref-4)