ABSTRACT

A graphical presentation package presenting the simulation of Hamming Codes, implemented as a project in computer graphics in the sixth semester B.E. (Computer Science & Engineering) curriculum of the Visvesvaraya Technological University, Belgaum. The aim of the project is to acquaint oneself with OpenGL functions, APIs and computer graphics algorithm.

Hamming code is named after R. W. Hamming of Bell Labs. It is a set of error-correction codes which are basically employed to detect and correct bit errors that can happen while moving or storing computer data. Similar to other error-correction codes, Hamming code brings into play the concept of parity and parity bits. Parity bits are the bits which are added to data for checking the validity of data, when it is read or after it has been received in a data transmission. Adding more than one parity bit, results not only in identification of a single bit error in the data unit, but also its location in the data unit

The main objective is, detection of single bit error, burst error and to correct by using the suitable Methodology (by hamming distance concept). The purpose of this project is to visualize unreliable Transmission of data during the transmission from encoder to decoder. It involves block coding, finding the hamming distance between to existing code words, redundancy, error correction, parity check code.

TABLE OF CONTENTS

1. INTRODUCTION

Computer Graphics is concerned with all aspects of producing pictures or images using a computer. A particular graphics software system called OpenGL, which has become widely accepted standard for developing graphic applications.

The application of computer graphics in some of the major areas is as follows:

- Display of information
- Design
- Simulation and animation
- User interfaces

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that we use to specify the objects and operations needed to produce interactive 3D applications.

Our project named “HAMMING CODES” uses OpenGL software interface. This project uses the techniques like input and interactions, menus and display list, transformations, colouring, texturing, shading, animations, etc.

1.1 INTRODUCTION TO OpenGL

OpenGL provides a set of commands to render a three dimensional scene. That means you provide the data in an OpenGL-useable form and OpenGL will show this data on the screen (render it). It is developed by many companies and it is free to use. You can develop OpenGL-applications without licensing.

OpenGL is a hardware- and system-independent interface. An OpenGL-application will work on every platform, as long as there is an installed implementation. Because it is system independent, there are no functions to create windows etc., but there are helper functions for each platform, a very useful thing is GLUT.

GLUT is a complete API written by Mark Kilgard which lets you create windows and handle the messages. It exists for several platforms, that means that a program which uses GLUT can be compiled on many platforms without (or at least with very few) changes in the code.

1.1.1 OpenGL Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. The following diagram (Figure 1.1) shows the Henry Ford assembly line approach, which OpenGL takes to processing data. Geometric data (vertices, lines, and polygons) follow the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer .

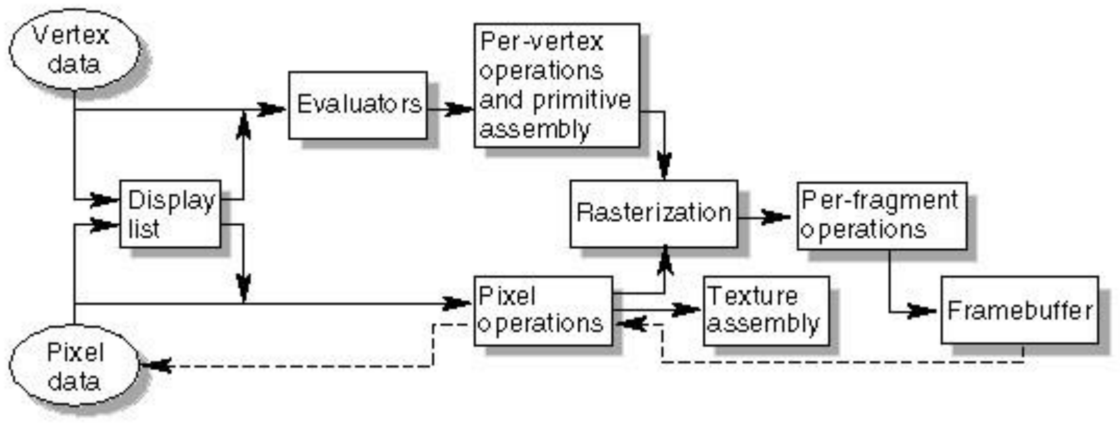


Figure 1.1 OpenGL Graphics Pipeline

Display Lists: All data, whether it describes geometry or pixels, can be saved in a display list for current or later use. When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.

Evaluators: All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points.

Per-Vertex Operations: For vertex data, next is the "per-vertex operations" stage, which converts the vertices into primitives. Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen.

Primitive Assembly: Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

Pixel Operations: While geometric data takes one path through the OpenGL rendering

pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory.

Texture Assembly: An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them.

Rasterization: Rasterization is the conversion of both geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the framebuffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support antialiasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

Fragment Operations: Before values are actually stored into the framebuffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled.

1.1.2 Working with OpenGL

OpenGL bases on the state variables. There are many values, for example the colour, that remain after being specified. That means, you can specify a colour once and draw several polygons, lines or whatever with this colour then. There are no classes like in DirectX. However, it is logically structured. Before we come to the commands themselves, here is another thing:

To be hardware independent, OpenGL provides its own data types. They all begin with "GL". For example GLfloat, GLint and so on. There are also many symbolic constants; they all begin with "GL_", like GL_POINTS, GL_POLYGON. Finally the commands have the prefix "gl" like glVertex3f (). There is a utility library called GLU, here the prefixes are "GLU_" and "glu". GLUT commands begin with "glut", it is the same for every library. You want to know which libraries coexist with the ones called before. There are libraries for every system, Windows has the wgl*-Functions, UNIX systems glx* and so on.

A very important thing is to know, that there are two important matrices, which affect the transformation from the 3d-world to the 2d-screen: The projection matrix and the modelview matrix. The projection matrix contains information, how a vertex – let's say a "point" in space – shall be mapped to the screen. This contains, whether the projection shall be isometric or from a perspective, how wide the field of view is and so on. Into the other matrix you put information, how the objects are moved, where the viewer is and so on.

OpenGL draws primitives—points, line segments, or polygons—subject to several selectable modes. You can control modes independently of each other; that is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the

frame buffer). Primitives are specified, modes are set, and other OpenGL operations are described by issuing commands in the form of function calls. Primitives are defined by a group of one or more vertices. A vertex defines a point, an endpoint of a line, or a corner of a polygon where two edges meet.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state querying commands return data that's consistent with complete execution of all previously issued OpenGL commands. The effects of OpenGL commands on the frame buffer are ultimately controlled by the window system that allocates frame buffer resources.

1.2BRIEF OUTLINE OF THE PROJECT

This project helps a user to understand the various stages of Hamming Code i.e Error Detection and Correction, which is widely used in the field of Computer Networks. This project uses various concepts OpenGL to display the various stages in the underlying process via diagrams.

1.3ORGANIZATION OF REST OF THE REPORT

LITERATURE SURVEY

This chapter gives a brief overview of the hamming codes.

Process, its advantages, disadvantages and applications.

 SYSTEM REQUIREMENT SPECIFICATION

This chapter specifies the various interface and functional requirements and broad system architecture.

 SYSTEM DESIGN

This chapter gives an overview of the architectural design and the various modules which are used for the implementation of the code.

 DETAILED DESIGN

It specifies the various Data Structures used in the code, along with the High Level Pseudo Code.

 TESTING

It explains the process of testing individual components and its working as an integrated system.

 CONCLUSION AND ENHANCEMENTS

This chapter specifies the final inferences made after the complete implementation of the project, learning its weaknesses and also understanding the additional features that could be added later.

2. LITERATURE SURVEY

Hamming codes are a family of linear error-correcting codes that generalize the Hamming (7, 4)-code, and were invented by Richard Hamming in 1950. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors.

2.1 ERROR DETECTION AND CORRECTION

Error Detection and Correction or **Error Control** are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channels are subject to channel noise, and thus errors may be introduced during transmission from the source to a receiver. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data in many cases.

Error detection : Error detection is the detection of errors caused by noise or other impairments during transmission from the transmitter to the receiver.

Error correction : Error correction is the detection of errors and reconstruction of the original, error-free data. This is shown in the figure 9.1

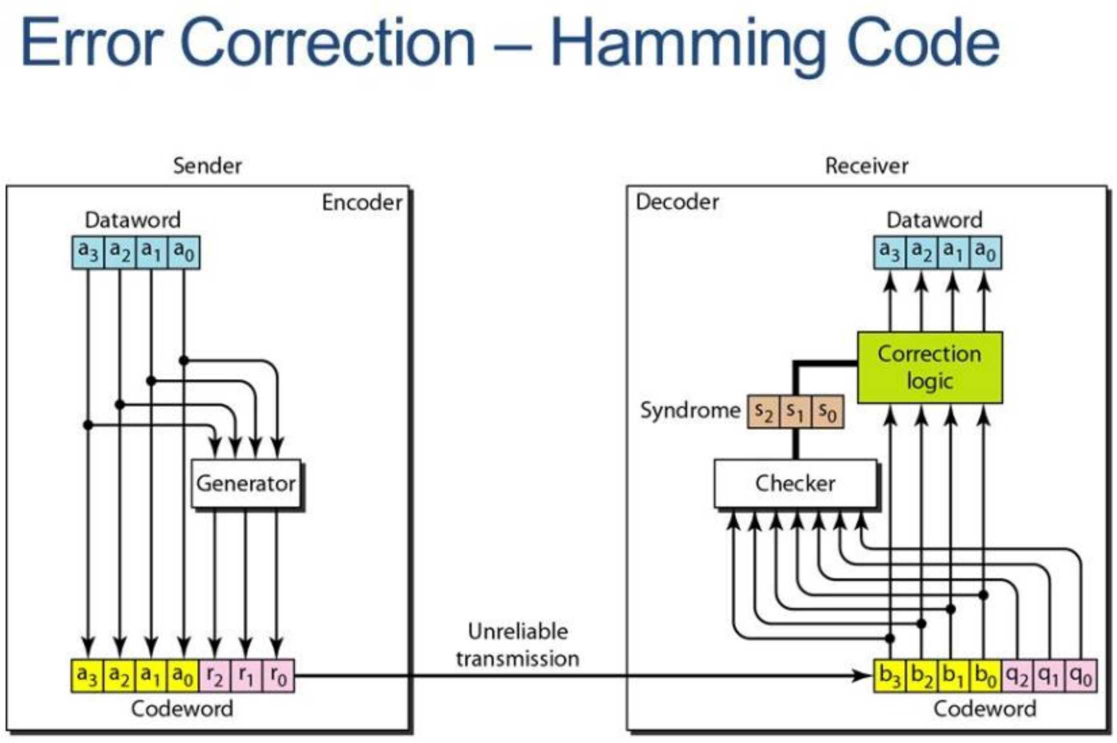


Fig 9.1 Hamming Code Block Diagram

2.2 ADVANTAGES OF HAMMING CODES

Hamming coders are preferred over the Single-Parity Check coders when detection as well as correction is desired.

- Hamming codes are capable of detecting up to two bits in error and have the ability to correct single bit errors. In particular, a single-error-correcting and double-error-detecting variant is generally referred to as SECDED.
- Hamming codes come in numerous varieties based on the number of message bits (m) and check bits (c) in its output.
- All Hamming codes are classified as linear block codes.
- The input to the Hamming coder is the m message bits and the output of the Hamming coder is an m+c codeword.
- The extra bits are added to the original data by a device which implements the generator matrix.
- The codeword is transmitted to the receiver where it is checked with another device which implements a parity check matrix. The output of the parity check device is a string of data bits with the same number of bits as the check bits. This output is called the Hamming syndrome.
- All the matrix multiplication is performed modulo-two.
- Hamming codes are identified by a set of ordered pairs which indicate their number of message bits and the number of bits in the codeword.
- For an (n,k) Hamming coder, k is the number of input bits and n is the number of output bits.
- The number of Hamming bits is determined by the number of data bits. The equation which determines the number of hamming bits is 2n-k n +1 where: k = number of data bits & n = total number of bits transmitted in a message stream.
- Hamming Code actually refers to a specific (7,4) code Hamming introduced in 1950. Hamming Code adds three additional check bits to every four data bits of the message. Hamming's (7,4) algorithm can correct any single-bit error, and detect all two-bit errors. Since the medium would have to be uselessly noisy for 2 out of 4 bits to be lost, Hamming's (7,4) is generally lossless.
- Because of the simplicity of Hamming codes, they are widely used in computer memory (RAM).

2.3 LIMITATION OF HAMMING CODES

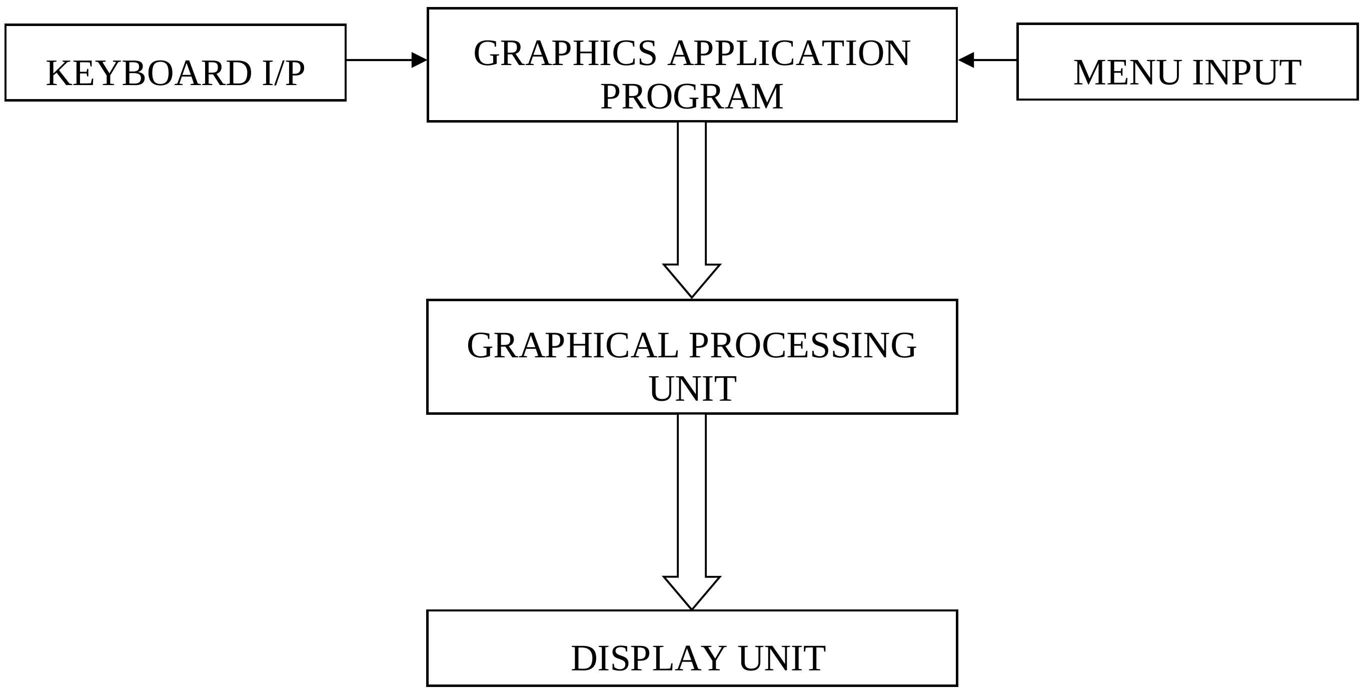
One of the main disadvantages of Hamming codes is the fixed Hamming distance and the problem of implementing coders for large blocks. The fixed Hamming distance of Hamming codes allows for detection of two error bits and the ability to correct only a single error bit. A code which enables the implementer to select the desired Hamming distance would be advantageous.

3. REQUIREMENT SPECIFICATIONS

3.1 EXTERNAL INTERFACE REQUIREMENTS

An easy way to use design has been chosen for user interface. The application interacts with the user by specifying options, and asking him/her to by press relevant **alphabetic keys** on the keyboard. The camera view is seen with the help of **right click** on the mouse.

3.2 BROAD ARCHITECTURE



4. SYSTEM DESIGN

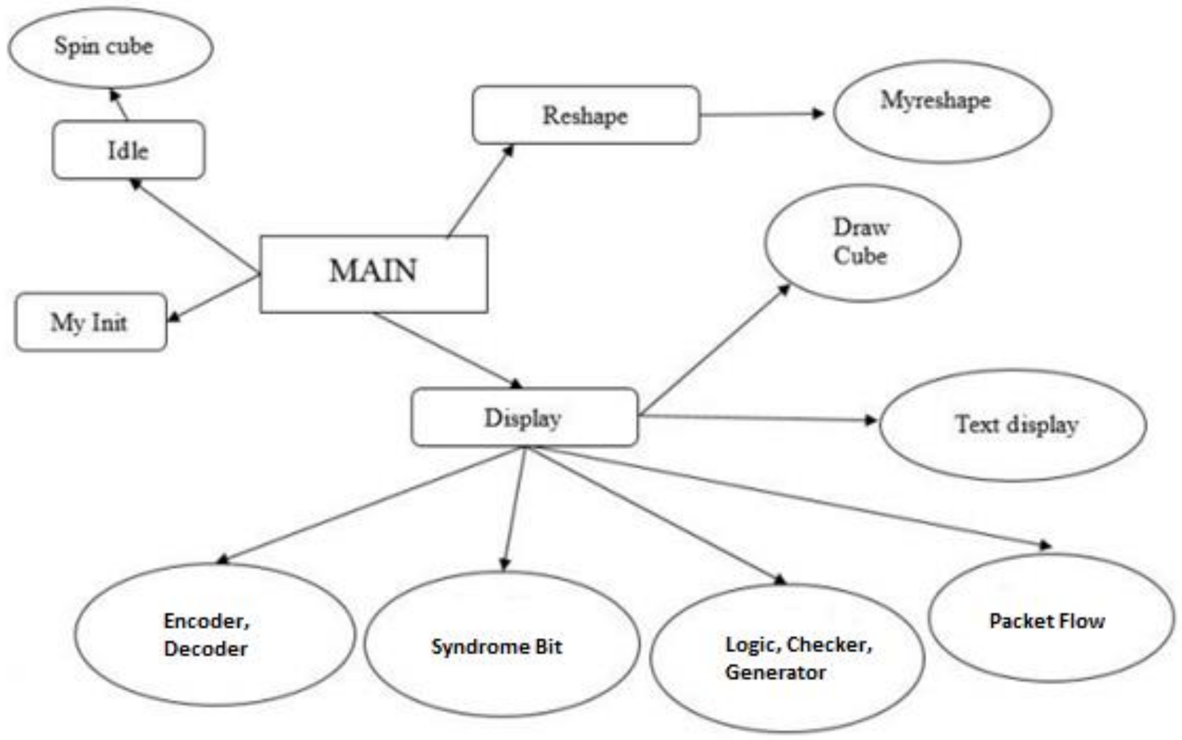
4.1 DESIGN OVERVIEW

Hardware delivers better performance but potentially longer development and less scope for change. Design both the hardware and the software associated with system. Partition functions to either hardware or software. Design decisions should be made on the basis on non-functional system requirements

4.2 SYSTEM ARCHITECTURAL DESIGN

The figure 4.1 shows the outline of the system architecture. How the modules call other modules.

Figure 4.1 - Architectural Design



4.3 COMPONENT DESIGN

This gives the brief outline of all the modules used in the project. This provides an efficient way to the developer for better understanding of the project.

- Main( )
  - Initialization of display mode
  - Creation of window along with size and position
  - Call for myInit( )
  - Call for callback functions
  - Enable hidden surface removal
  - Call for Event loop
- myReshape( )
  - Sets the viewport on changing the shape of the window
- Mouse( )
  - Used for performing functions on the left and right buttons on the mouse
- Keyboard( )
  - Used for performing functions on pressing different keys on the keyboard.
- SpinDisplay( )
  - Sets the spin of the rectangular boxes.
- select( )
  - Used for displaying the text strings.
- display( )
  - Used for displaying content on the window.
- boxes( )
  - Used to draw the Encoder and Decoder Boxes.
- logic( )
  - Built for Correction Logic.
- checker( )
  - Built for Checker.

- generator( )
    - Built for Generator.

- syndrome( )
    - For forming Syndrome Bits.

- decoder()
    - For forming Decoder.

- encoder()
    - For forming Encoder.

- packet( )
    - Packet Movements for understanding data flow.

- lines( )
    - Used for lines connecting different devices.

- all_text()
    - Function used to print all the text.

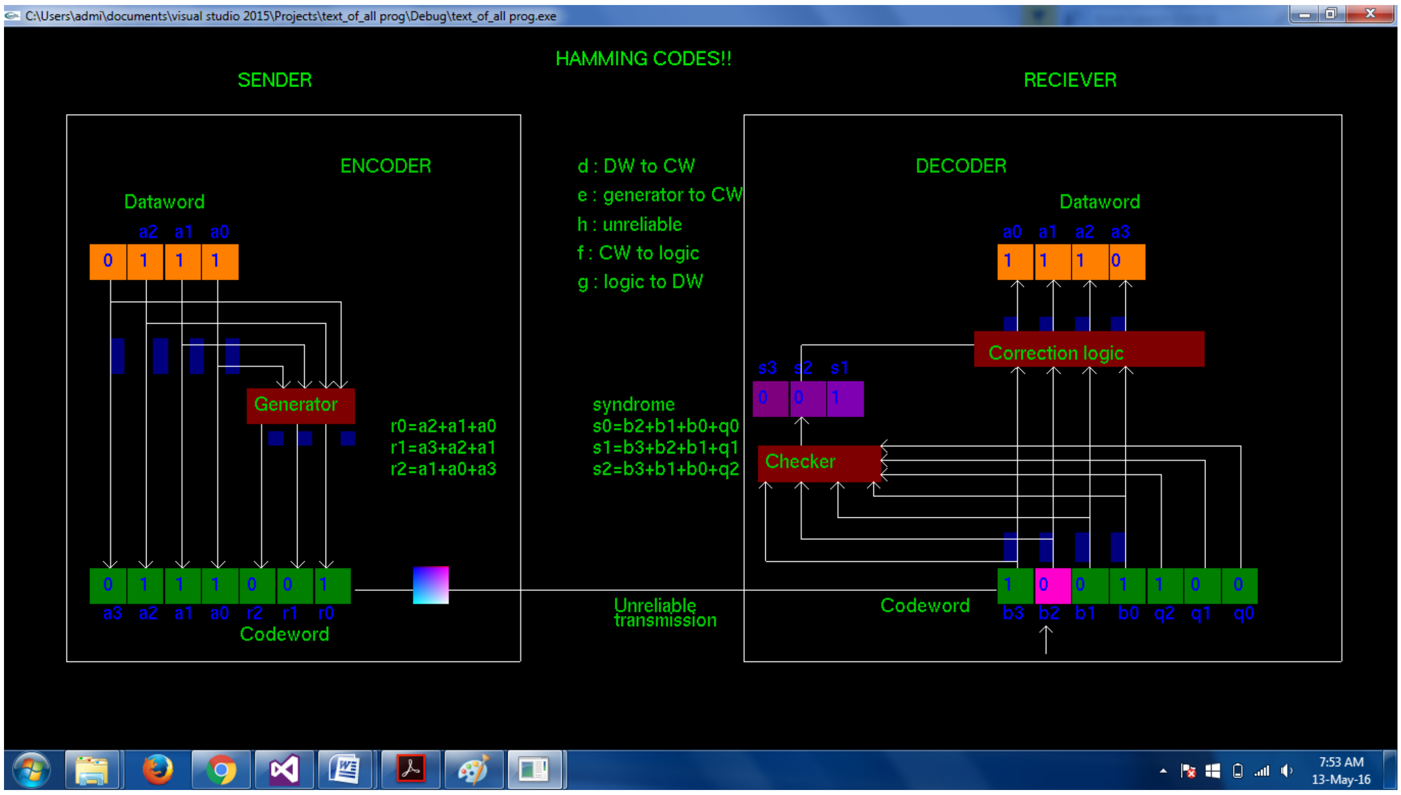
- rect( )
    - Function used to form 2D rectangle.

- colorcube()
    - Function for unreliable transformation cube.

- polygon()
    - Function to form faces of the cube.

- rot()
    - Function for camera movement.

4.4 USER INTERFACE DESIGN



5. DETAILED DESIGN

5.1 DATA STRUCTURES

5.1.1 Data Definitions

IDENTIFIER TYPE DESCRIPTION

- vert[][4] float - Defines the vertices of the cube
- color[][4] float - Defines the colors used in the application
- theta[] float - Define the rotation angle
- axis int - Defines the axis of rotation
- First int - Save the return value after creating first window
- Sec int - Save the return value after creating second window
- Third int - Save the return value after creating third window
- Fourth int - Save the return value after creating fourth window
- miny int - Defines the starting point of y-axis
- minx int - Defines the starting point of x-axis
- count int - Stores the no. of samples collected
- max float - Defines the max time until which the graph has to be plotted
- inc float - Defines the increment in time
- x float - Defines the time(t) value
- y float -Defines the amplitude(a) value
- f float - Defines the frequency of the input curve
- r float - Defines the resolution of the graph (inverse)
- s float - Defines the scale of the graph
- sr float - Defines the sampling rate
- sv[8] float - Stores the sampled values
- nqv[100] float - Stores the quantized values
- t[3] float - Temporary variable to store the rotation angle

CONCEPTS USED

- ➢ Translation
- ➢ Animation
- ➢ Camera movement
- ➢ Menu
- ➢ Keyboard
- ➢ Mouse
- ➢ Shading

5.2 TOP LEVEL PSEUDO CODE

1. Algorithm: select

```
Input: The coordinates where text has to displayed, color, font style and pointer to the text array
Method:
Set the text color
Set the raster position
Repeat
Display character
Until end of text
```

2. Algorithm: polygon

```
Input: void
Method:
Begin GL_QUADS
Repeat
Specify the face color
Specify vertices of the corners of the face
Until all faces are drawn
```

3. Algorithm: colorcube

```
Input: 4 vertices  of a polygon
Method:
Call the function polygon by passing 4 vertices
Repeat it for all the 6 faces of polygon.
```

4. Algorithm: rot

```
Input: void
Method:
Set the axis of rotation axis
Repeat
Rotate the frame
```

5. Algorithm: spindisplay1

```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
```

6. Algorithm: spindisplay2

```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
```

7. Algorithm: spindisplay3
```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
```

8. Algorithm: spindisplay4
```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
```

9. Algorithm: spindisplay5
```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
```

10. Algorithm: spindisplay6
```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
```

11. Algorithm: spindisplay7
```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
```

12. Algorithm: spindisplay8
```
Input: void
Method:
Set the axis of rotation as z axis
Repeat
Rotate the frame
Swap Buffers
Until angle becomes equal to the maximum defined rotation angle
```

13. Algorithm: print1
```
Input: void
Method:
Define the various texts to be displayed
Display the college logo
Display texts by calling display text function
Flush the output on the screen
```

14. Algorithm: print2

```
Input: void
Method:
Define the various texts to be displayed
Display the college logo
Display texts by calling display text function
Flush the output on the screen
```

15. Algorithm: display

```
Input: void
Method:
Define the texts to be displayed in arrays
Clear the color and depth buffers
Set the rotation references
Flush the output on screen
Clear buffers
Flush the output on screen
Call the spin function
Call display0
```

16. Algorithm: myReshape

```
Input: void
Method:
Set the viewport
Set the matrix mode to GL_PROJECTION
Set the clipping coordinates using glOrtho
Set the matrix mode to GL_MODELVIEW
```

17. Algorithm: mouse

```
Input: The coordinate where mouse clicked and button clicked along with the state
Method:
Ignore
```

18. Algorithm: keyboard

```
Input: The key pressed
Method:
Destroy the first window
Reset the reference axis for new window
Reset the reference angles and flag values for next window
Create appropriate window
Create menu entry wherever required
Attach menu to GLUT_RIGHT_BUTTON
Register appropriate reshape function
Register appropriate keyboard function
Register appropriate display function
Call init
Enable depth test
Run main looped infinite no. of times
Register appropriate display function
Call init
Enable depth test
Run main looped infinite no. of times
```

19. Algorithm: main

```
Input: void
Method:
Set the display mode to GLUT_DOUBLE, GLUT_RGB, GLUT_DEPTH
Create the welcome window
Register appropriate reshape function
Register appropriate keyboard function
Register appropriate display function
Call init
Enable depth test
Run main looped infinite no. of times
```

OpenGL Commands Familiarized:

- glRotatef
- glMatrixMode
- glEnable
- glutSwapBuffer
- glutIdleFunc
- glNormal
- glViewport
- glutReshapeFunc
- glutMouseFunc
- glutKeyboardFunc

6. TESTING

6.1 TEST PLAN

The testing is done phase by phase initially. For testing Purpose the System was considered to be having 4 events/units.

The units are as follows:

1) Displaying the Welcome Screen with animated text menu.

2) Displaying the animated Hamming Codes in a new window.

3) Working of data.

4) Displaying rotation in the same window.

Each of these units is further sub divided into sub units. But they would be looked into, if and only if there's an error found in the top level implementation of these units while testing.

6.1.1 Unit Testing

Testing of Unit 1:

- The program is then run to see if it produces the desired result
- Check whether the Window is of the Desired properties
- If the scene rendered is similar to the scene we had imagined then it's fine, if not the following tests are further made into the subunits of unit 1
  - ✓The contents of the display function are checked.
  - ✓ If the scene isn't visible then the camera position and the viewing cube are checked.
  - ✓ If the scene is way too much zoomed or way too far then the viewer location is altered.
  - ✓ If any object isn't being rendered then its corresponding rendering function is further put through testing.
- If all the above tests are passed then we can move on to the next unit testing.

Testing of Unit 2:

- This program component is run to see if it produces the desired result
- If the output display an animation which has text moving upwards then the output is as desired or else,
  - ✓ Check the display function if all the values are properly defined or not.
  - ✓ Check increment value if it as per requirement or not. If not satisfactory then adjust it.
  - ✓ Check if buffers are cleared at the starting of every iteration or not.
- If all the above tests are passed then we move on to the next unit testing.

Testing of Unit 3:

- This program component is run to see if it produces the desired result
- If the output displays an text to choose input wave configuration using mouse right clicks.
- Each component of menu is checked whether it is functioning as desired or not.
- Then check whether the display after the input is rendered as desired or not.
    - ✓ Check the menu function for errors. Rectify the errors, if any.
    - ✓ Check the rotation function of cube and also the various plotter functions.
    - ✓ Check if SwapBuffer() has been given at the end of function or not.
- If the graphs have an error, check whether the values displayed on the console are correct or, if correct then check the graph parameters and plot_pixel functions.
- If all the above tests are passed then we move on to the next unit testing.

Testing of unit 4:

- If the output display an animation which has text moving upwards then the output is as desired or else,
    - ✓ Check the display function if all the values are properly defined or not.
    - ✓ Check increment value if it as per requirement or not. If not satisfactory then adjust it.
    - ✓ Check if buffers are cleared at the starting of every iteration or not.
- If all the above tests are passed then we move on to the next unit testing.

If all the above tests are passed then we move on to System Testing

6.1.2 System Testing

Once the entire unit testing is done then the final test phase has reached. This is the System Test phase where the entire system is checked in a single go. There are many times when the units work properly independently but when brought together to form a system and then tested many a times uncertain outputs are found. Hence this is very important.

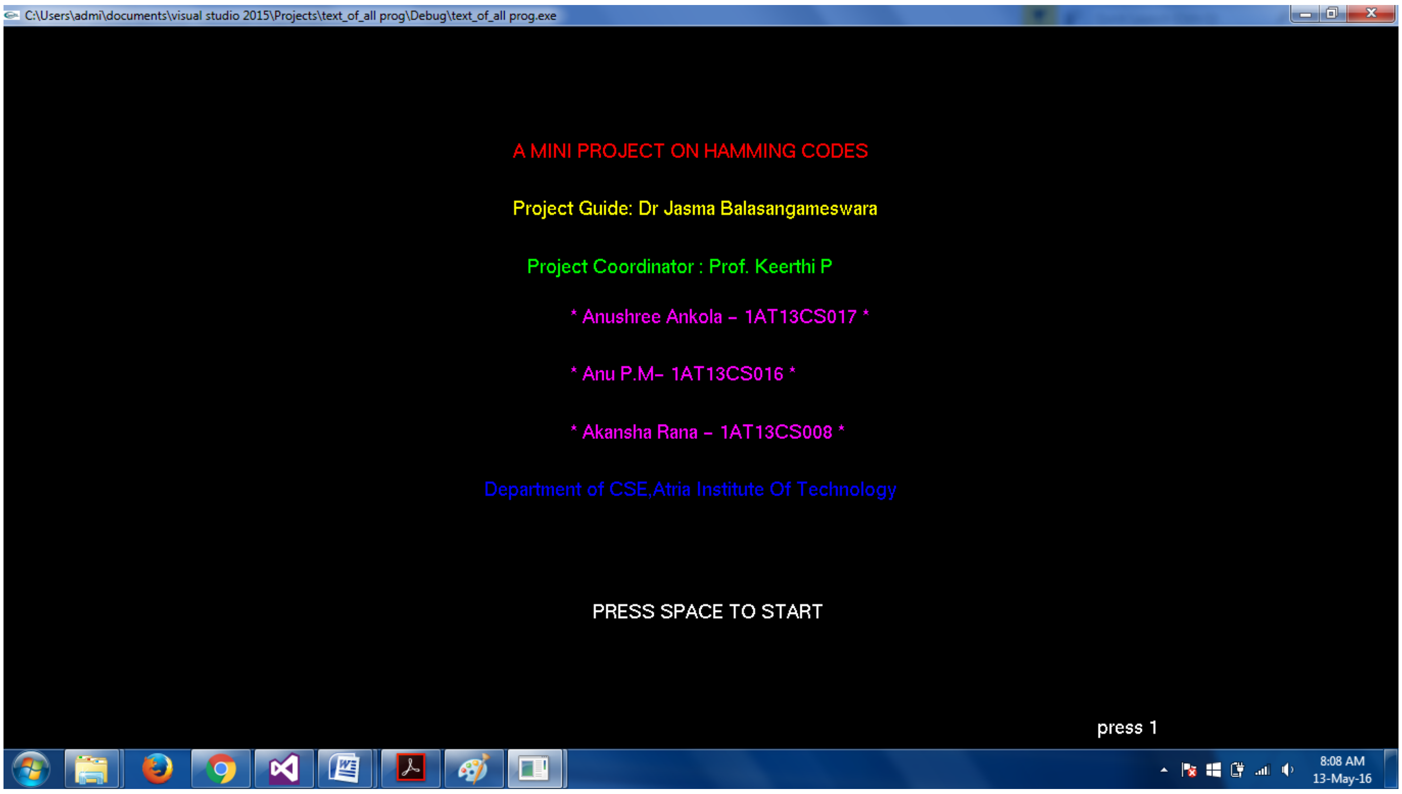
The testing process followed is as follows:

- Check for any compile time or Syntax error.
- Check whether the window is created as desired, without error.
- Check whether the cube is displayed as per the specification.
- Check whether the graphs plotted are as desired or not.
- Check whether all the menu components are working or not.
- Ensure that switching between windows is smooth and error free.
- Check whether the animation is consistent or not.
- Check whether the outputs of various phases are in compliance with the input wave configurations or not.

Our Graphics Application passed through all of the tests stated above and hence we can say that our project is a success. Test Results/Snapshots are given in the next section.

6.2 TEST RESULTS/SNAPSHOTS

Fig. 6.1.: Testing of Welcome Screen's animation



C:\Users\admi\documents\visual studio 2015\Projects\text_of_all prog\Debug\text_of_all prog.exe

A MINI PROJECT ON HAMMING CODES

Project Guide: Dr Jasma Balasangameswara

Project Coordinator : Prof. Keerthi P

* Anushree Ankola – 1AT13CS017 *

* Anu P.M– 1AT13CS016 *

* Akansha Rana – 1AT13CS008 *

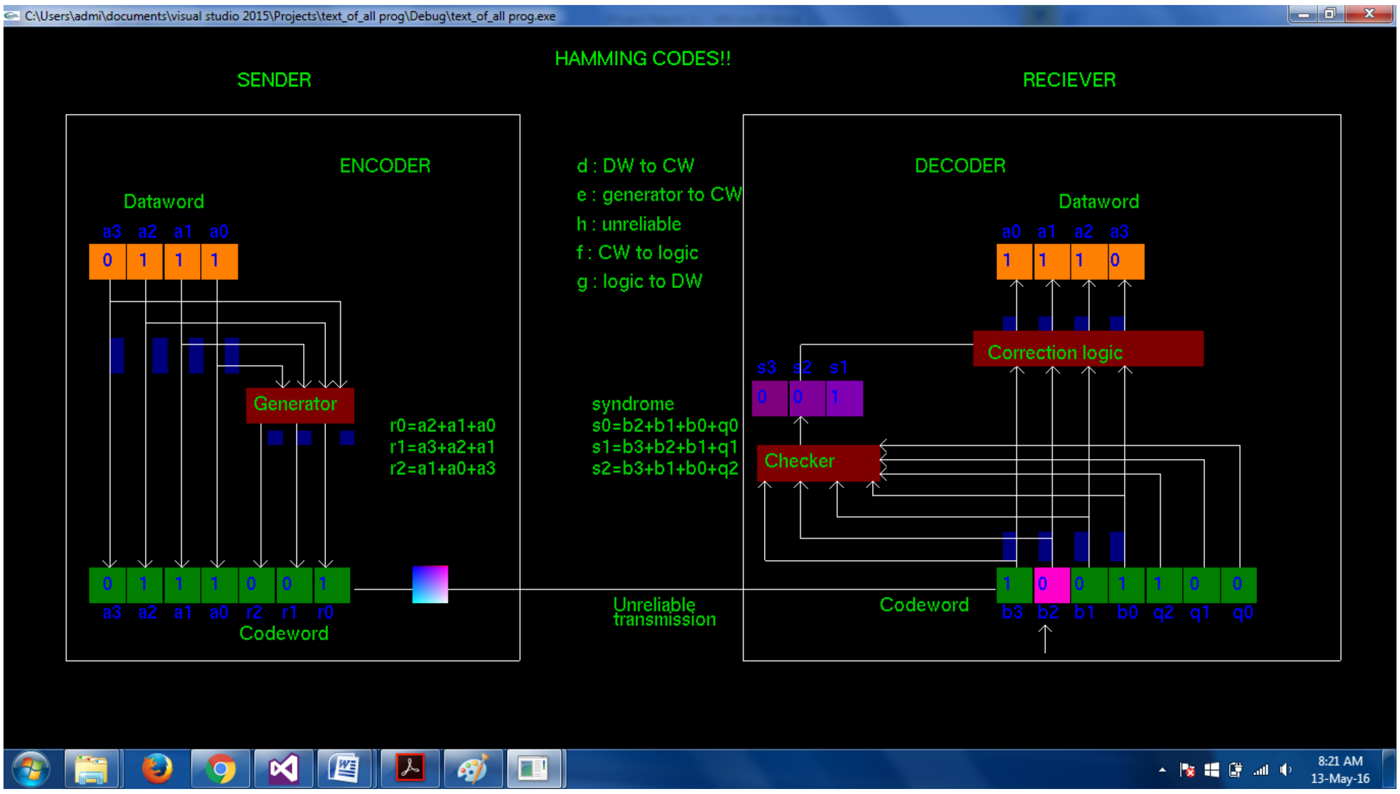
Department of CSE,Atria Institute Of Technology

PRESS SPACE TO START

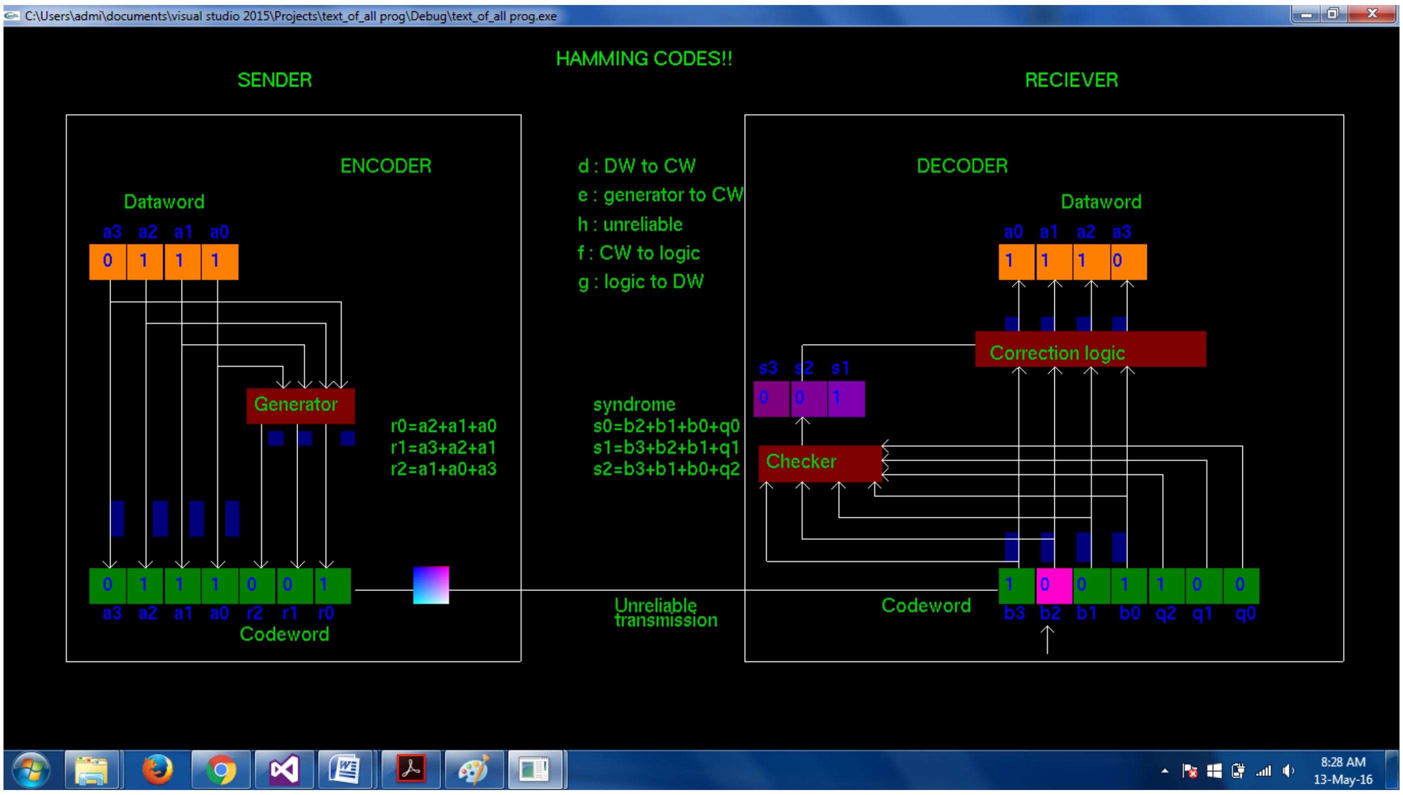
press 1

8:08 AM 13-May-16

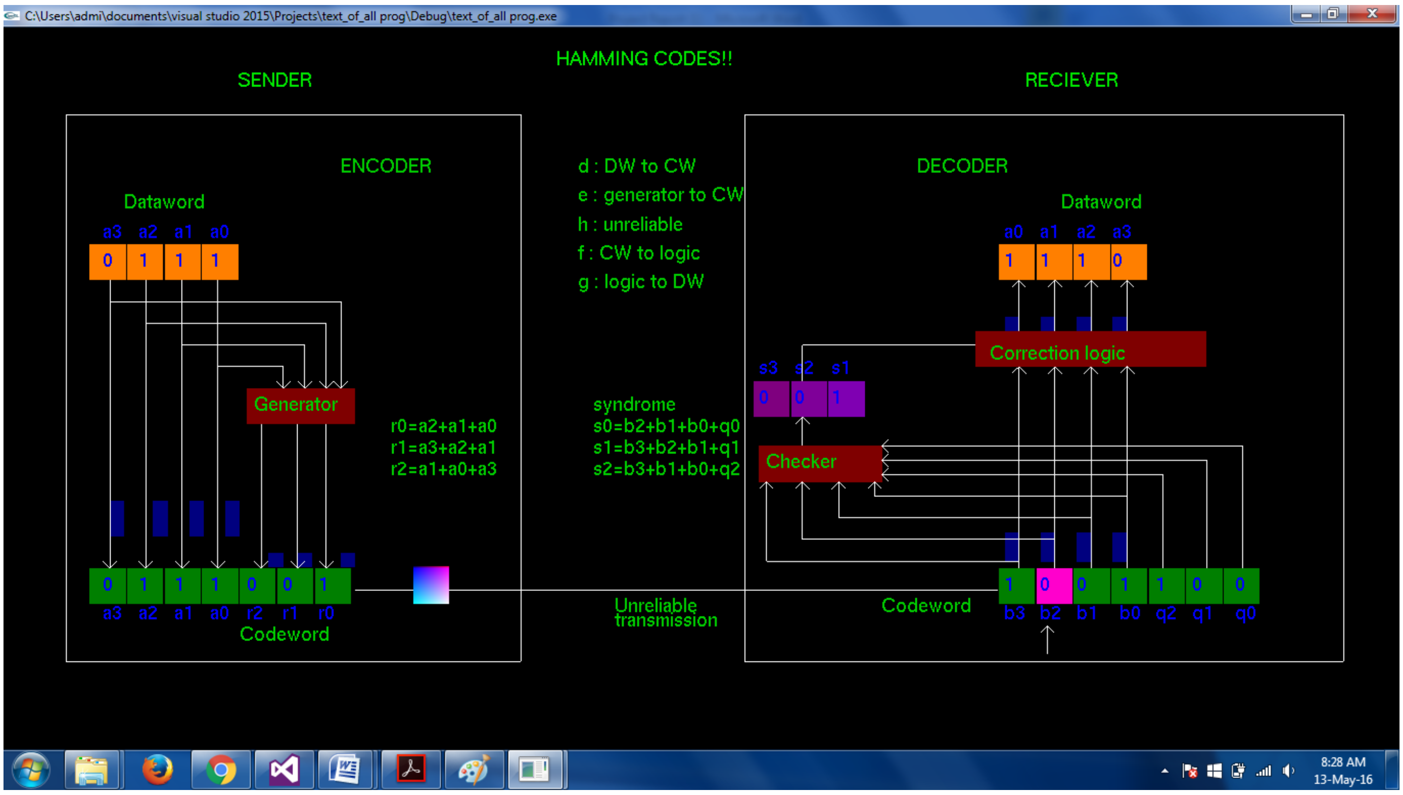
6.2 Testing of Dataword and Codeword Animation



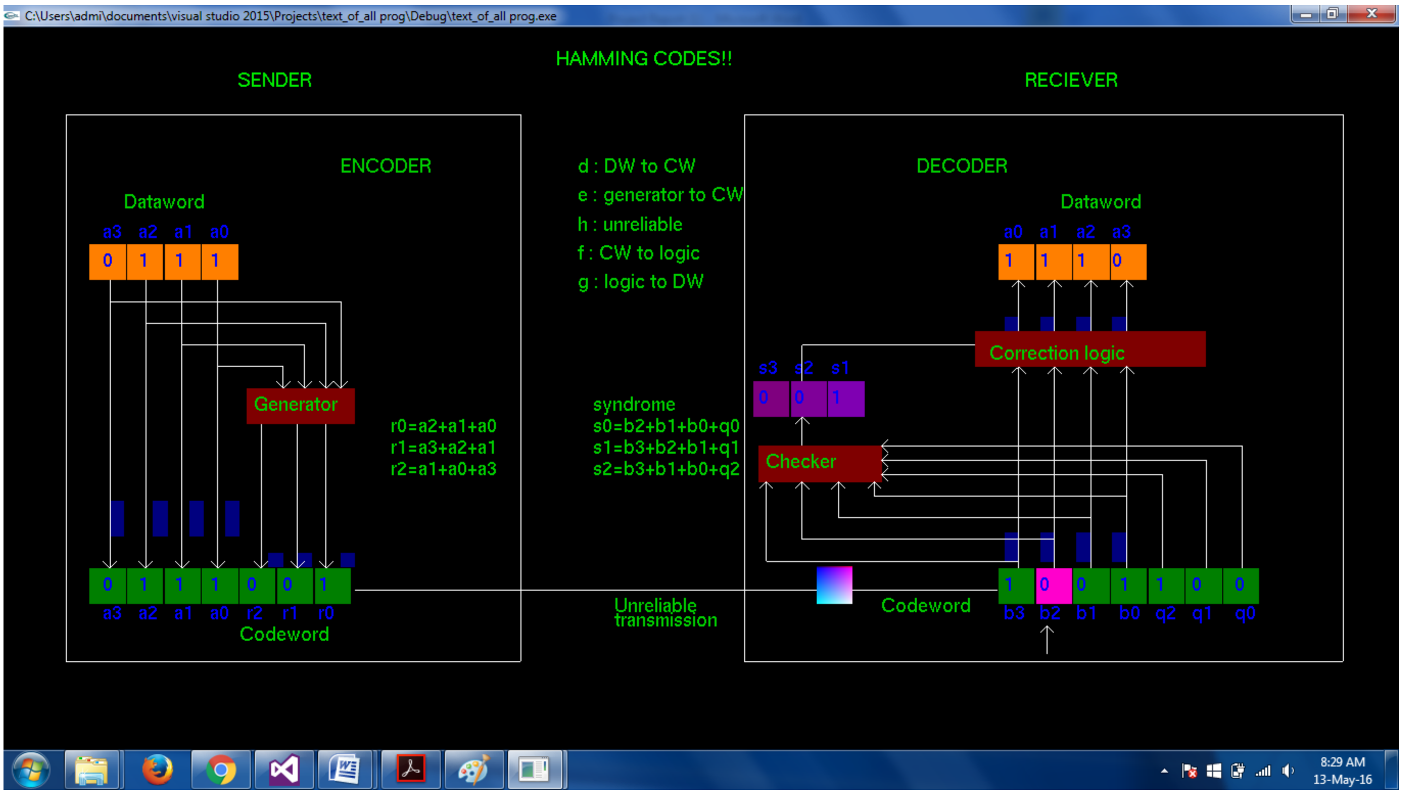
6.3 Testing of Packet Movement from Dataword to the Codeword by pressing Key ‘d’



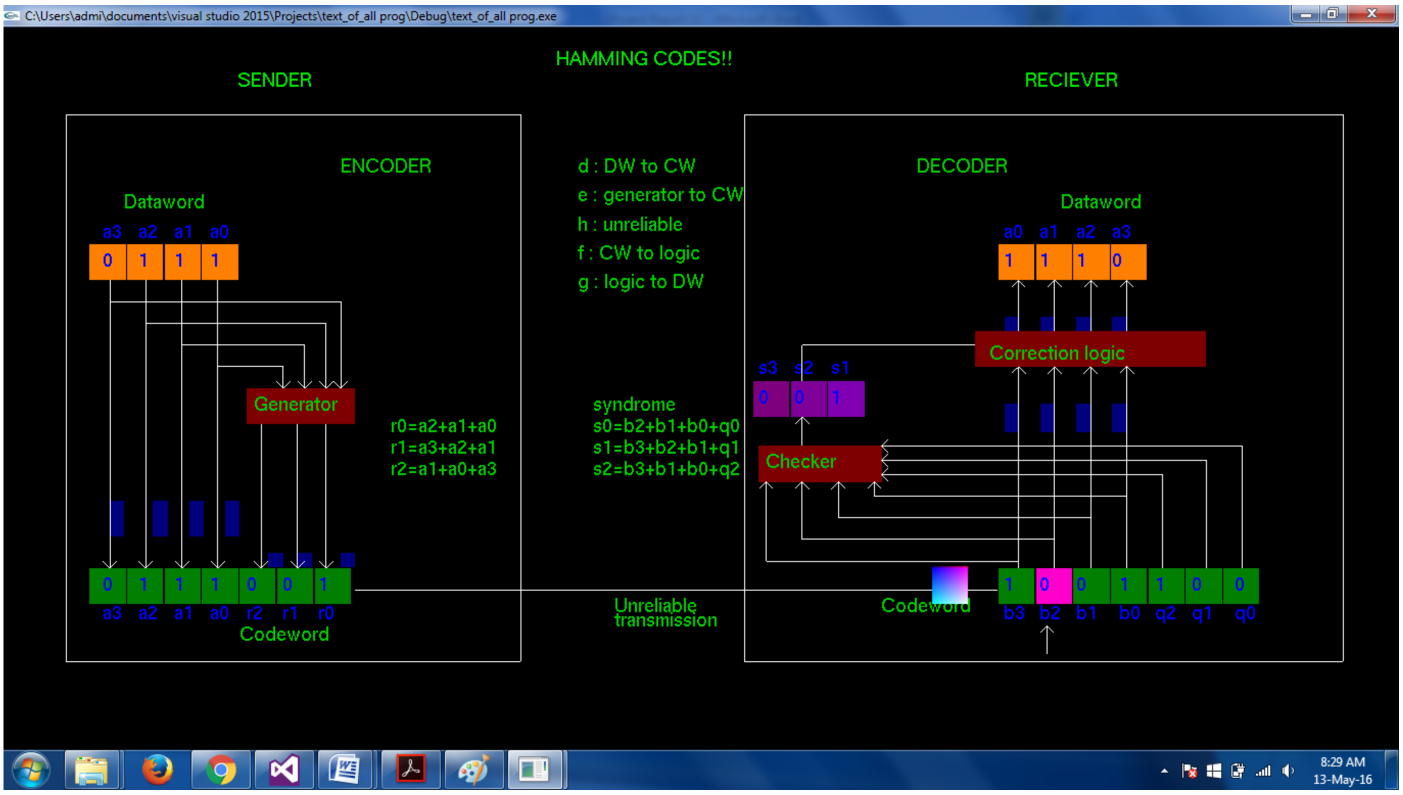
6.4 Testing of Packet Movement from Generator to the Codeword by pressing Key ‘e’



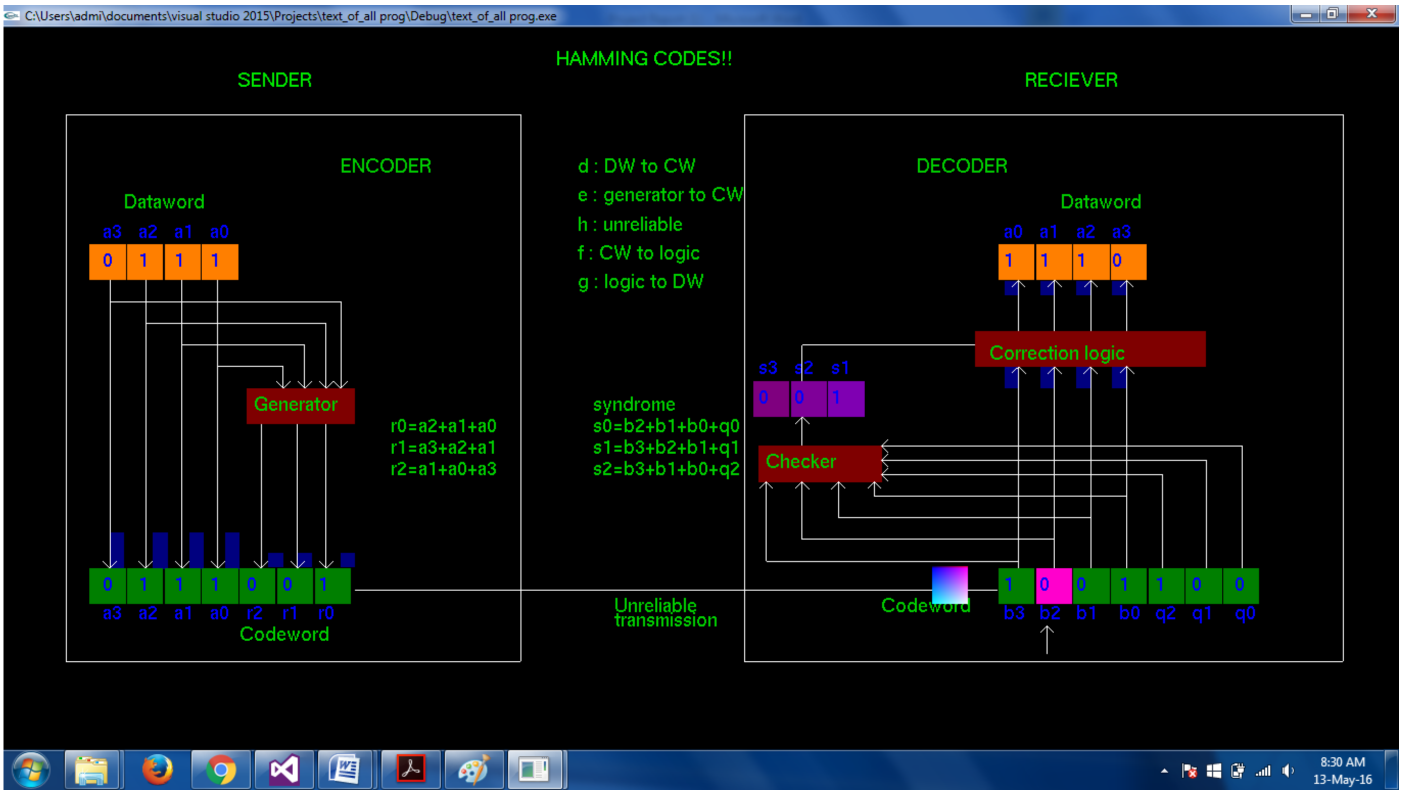
6.5 Testing of Shaded Packet's Unreliable Transmission by pressing Key 'h' and error bit identification



6.6 Testing of Packet Movement from Codeword to the Correction Logic by pressing Key 'f'



6.7 Testing of Packet Movement from Correction Logic to the Dataword by pressing Key ‘g’

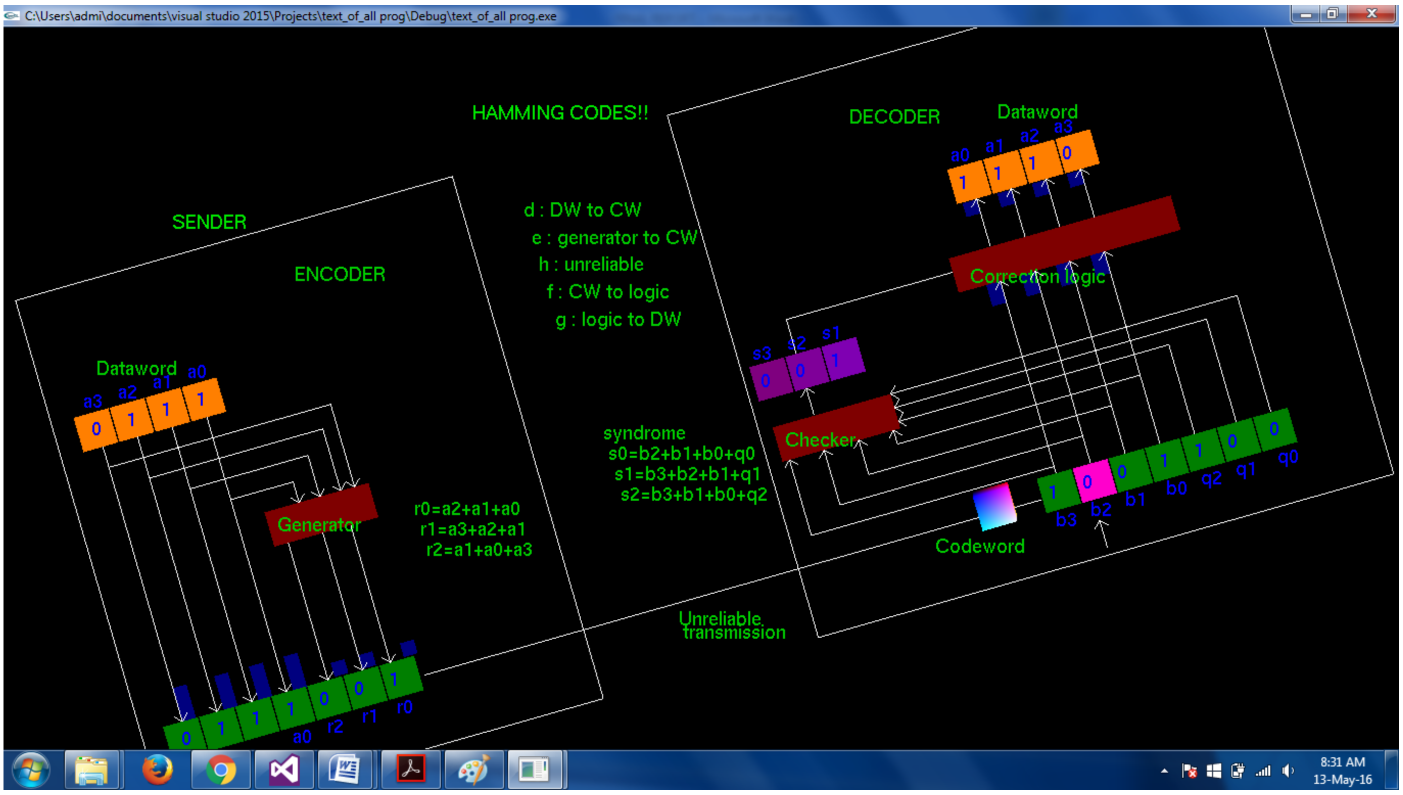


d : DW to CW
e : generator to CW
h : unreliable
f : CW to logic
g : logic to DW

r0=a2+a1+a0
r1=a3+a2+a1
r2=a1+a0+a3

syndrome
s0=b2+b1+b0+q0
s1=b3+b2+b1+q1
s2=b3+b1+b0+q2

6.8 Testing the Camera View1



d : DW to CW
e : generator to CW
h : unreliable
f : CW to logic
g : logic to DW

r0=a2+a1+a0
r1=a3+a2+a1
r2=a1+a0+a3

syndrome
s0=b2+b1+b0+q0
s1=b3+b2+b1+q1
s2=b3+b1+b0+q2

7. CONCLUSION AND SCOPE FOR FURTHER ENHANCEMENTS

We had a great experience in the course of designing this project, which made us discover and learn many things that pertain to the topic of OpenGL programming. We have tried to our best potential to incorporate all the basic level requirements of a normal graphics package for Windows operating system.

The aim of the project is to visually run the implementation of Hamming Codes using OpenGL APIs. This can be further enhanced to provide better understanding of the Hamming Code technique.

The following improvements can be made in future:

- Add Lighting to correctly understand where the error is being detected.
- Computation at the Generator to obtain bits r0,r1,r2.
- Computation at the Checker to find the syndrome bits.
- Blinking effect of the error detected bit in the receiver or the decoder.
- Add sound features to it so as to provide a better understanding of the concept.

8. APPENDIX

8.1APPENDIX-I

8.1.1 Bibliography

8.1.1.1 Reference Books/Papers

- Interactive Computer Graphics: A Top-Down Approach using OpenGL (5th Edition) By Edward Angel, Pearson Publication
- Data Communication and Networking (4th Edition) By Behrouz A Forouzan, McGraw Hill Education

8.1.1.2 Web Sites

- http://www.opengl.org
- http://elearning.vtu.ac.in
- http://www.w3schools.com
- http://www.glprogramming.com/red

8.2 APPENDIX-II

8.2.1 Development Tools

- Microsoft Visual Studio 2010 Ultimate Edition

8.2.2 Software Environment

- Microsoft Windows 7 Home Premium
- GLUT 3.7.6

8.2.3 Hardware Environment

System	
Manufacturer	Dell
Model	Dell Inspiron 1440
Total amount of system memory	3.00 GB RAM
System type	32-bit Operating System
Processor	Intel® Core™ 2Duo CPU T6600 2.20GHz
Storage	
Total size of hard disk	320 GB
Disk partition (C:)	11 GB Free (78 GB Total)
Disk partition (D:)	03 GB Free (15 GB Total)
Disk partition (E:)	28 GB Free (205 GB Total)
Graphics	
Display adapter type	Intel® GMA 4500 MHD
Total available graphics memory	1695 MB
Dedicated graphics memory	32 MB
Dedicated system memory	32 MB
Shared system memory	1631 MB
Display adapter driver version	8.15.10.1994
Primary monitor resolution	1366x768