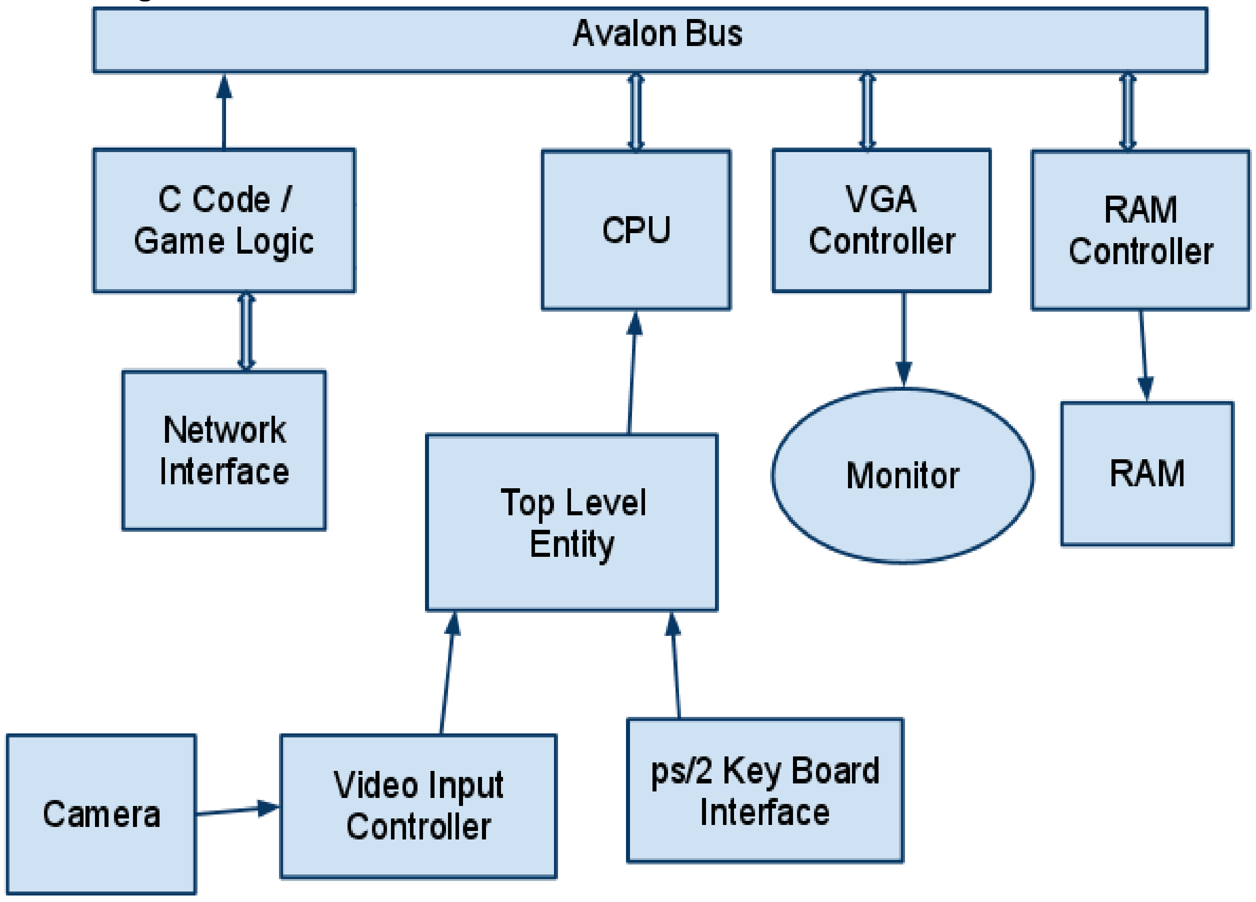
SAGa

Carlos Abad - caa2140
Matthew Duane - md2835
Anthony Garvan - ag3035
Sam Friedman - snf2108

Introduction

Our team will design and implement a Sprite Animated Game (SAGa) modeled after the turn-based shooter Worms. The game will feature sprite-based characters moving about a gameboard, and will include a variety of sound effects and animations, and will be displayed on the connected monitor at 640X480 resolution. Gameplay will be two players simultaneously, either on the same computer or via a network connection. Keeping in line with the Worms gameplay, our game will feature multiple sprite characters per team (the original game typically supports 4 per side), with each being allowed to move and attack the opposition on a turn-by-turn basis. Movement of sprites will be in both the X and Y frames. Multiple weapons and location-specific damage will also be included, but at this point specific levels for each are unknown.

Basic Diagram



I. VGA Controller/Sprite Engine

The resolution for the VGA monitor is 640X480, so for simplicity we will make sure any pre-rendered Characters and Tiles are proportional to this measurement. Thus, both Tiles and Character sprites will be displayed in 16X8 pixel dimensions. That will make it easier for us to transmit exact locations both between memory and the VGA controller as well as through the network as packets.

Sprite Creation and Display

As we learned in the Sprites lecture, it is more efficient to store and draw pre-rendered Characters and Tiles as opposed to drawing them in real-time. The entire gameboard will be stored in RAM as a 40X30 array, with a 1 byte value specifying the type of sprite or Tile (or lack thereof). Table 1.1 specifies the individual code and representative sprite.

Value	Sprite/Tile Type
0	None/transparent
1	Worm/player
2	Lava tile
3	Ground tile
4	Sky tile
5	Weapon/bullet

Table 1.1: Sprite Array Codes

Since the character and weapon sprites can be placed arbitrarily on this grid, we will use the pixel location to “map” its location to the grid and display the given sprite at that location.

Here are some example Sprites and Tiles that we plan to use.

Sprites from http://www.yoda.arachsys.com/worms/wa/anims/

The actual sprites used in the game will be rendered in multiple “sprite arrays.” Tiles and Character sprites will be afforded 4 colors, meaning each pixel location for that Sprite will have a 3-bit “color code” to specify which color should be visible (or if that bit should be transparent). By using 3 bits, this affords us some flexibility if we would like to add additional color depth for a particular sprite.

II. Keyboard Inputs

The PS2 Keyboard will be used for input from the user.

Characters will only be able to move in the X plane, meaning the ⇠ and ⇢ arrow keys will be used for character locomotion. Pressing a direction opposite of the direction the character is facing will turn the character around.

Aiming for a character is achieved by rotating the “sight” of the gun up to 90º above and below the X axis. Pressing the ⇡ arrow increases the angle while pressing the ⇣ arrow decreases the angle.

III. RAM Controller and Memory-Mapped I/O

We expect to utilize both the FPGA's on-board memory as well as the 512K SRAM module utilized in Lab 3. At this time, we expect to store all of the sprites and tiles in the onboard memory, since they are small and immutable.

Characters and Tiles

The size of a 3-bit Character sprite will be 1024 bytes (16X8 bytes * 8 [3-bit color code]). Depending on implementation, there will either be a single Character for all players (the only difference being the direction it is facing), or two Characters with a distinguishing characteristic (such as a different body color). With 4 unique Weapon projectiles, that increases the number of potential characters to 6, meaning 6KB used. Factoring in 8 unique Tile types increases the total number to 14 KB, still well below the 62KB found on the FPGA.

Gameboard and Audio

Since both the gameboard and audio components of the game need not be accessed as quickly nor updated as frequently as the sprites, we expect to store them on the board's SRAM module. The 16-bit bus width and slower response times for reads and writes will be issues, but we them to be less pressing because both elements are time-lenient, as the gameboard would only be altered at the end of a turn and the audio sound effects would be needed infrequently and only at specific instances (e.g. player being hit by a projective or dying).

Memory-Mapped I/O

Though subject to change, here are the know memory mappings for the peripherals we expect to use in the project:

Peripheral Name	Mapping Size
Keyboard	8 bytes
Audio	8 bytes
Jtag_UART	8 bytes
SRAM	524,288 bytes

IV. Network Interface

The network interface for SAGa will be similar to the paradigm utilized in Lab 2, meaning the device will utilize standard UDP packets with IP headers specifically designed for this application and the ad hoc network that exists in the lab.

A standard Ethernet packet features 4 “layers” that can be accessed programmatically by the user, with each layer incorporating header information and/or data specific to the routing of that packet over a network. These four layers and associated data are:

Layer	**Data**
Link	Source and Destination MAC addresses plus Packet Type
Network	IP Header (defined below)
Transport	UDP datagram (defined below)
Application	SAGa-specific protocol (defined below)

Network Layer

The **Network Layer** includes the IP Header, which is required for determining how and where to send a packet between two machines. It includes a number of pre-determined fields along with optional fields, all of which are defined below:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Version = IPv4				Packet header length				Differentiated services								Total datagram length															
Packet ID																Frag Flags (0)				Fragment offset (0)											
Time to live (i.e. packet life)								Packet protocol (UDP is x11)								Header Checksum															
Source IP Address																															
Destination IP Address																															

Due to the minimal amount of data that should need to be transmitted between the two computers, we do not foresee the need for multiple packets per transaction, nor do we expect any fragmentation of datagrams to occur because the maximum transmission unit (MTU) for the network is larger (1500 bytes) than the expected maximum packet size (1024 bytes). These assumptions eliminate the need to contend with the fragmentation flags and offsets found in a typical IP header. If during implementation these assumptions prove to be incorrect, then we will implement the necessary fragmentation procedures for packet transmission.

Transport Layer

The **Transport Layer** features the actual UDP datagram, which includes both the UDP header and the data to be used by the Application Layer. The UDP header is 8 bytes long and includes the following information:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Source port #																Destination port #															
UDP Packet length																Checksum															

Both the source and destination ports can be set to some generic value because the packets will only be traversing the network in the lab. If this application was ported to communicate between computers on different networks, these values would need to be coincide with ports that are open on both networks (such as 80 for HTTP and 25 for SMTP) and be consistent across all systems.

The UDP packet length will be computed by combining the size of the UDP header (8 bytes) with the total number of bytes in the UDP payload. Though the maximum possible UDP payload is 8184 bytes (8192 bytes – 8 byte header), our maximum UDP packet size will be 512 bytes, with 8 bytes dedicated to the header.

The UDP checksum is not required for IPv4 packets (RFC 768), and thus will be sent to 0 since any error correction will occur in SAGa and not by the network.

Application Layer

The **Application Layer** represents the protocol utilized by SAGa to transmit information about the current game state between networked computers. There are three game events that will lead to the transmission of data between the two clients:

1. **Player movement** - The active player changes its position on the gameboard. The packet will include the player's current location (x and y coordinates) on the gameboard.
2. **Player Fires Weapon** - The active player fires its weapon. The packet will include the player's current location (x and y coordinates) on the gameboard, the angle at which the player fired, the "power" of the shot (as defined below in the **Gameplay** section), and optional additional data such as the type of weapon fired (if different than the default).
3. **Turn Completion** - Sent by the active player's computer upon the completion of a turn. In addition to signifying the completion of a turn and shift control to the other user, this packet will serve as a "sanity check" for the game, including data for standardizing the game environment in light of the last turn. The data included in this packet will include, but is not limited to:
    a. The location of each player.
    b. The health of each player.
    c. Any Tiles that were affected during the last turn. This will be treated as a simple on/off switch - if there was a Tile at this location previously, it will be removed; if not, a Tile of specified type will be added.

In the event that the non-active player's game state does not match this data, it will treat this data as correct and alter its state accordingly. By following this protocol, we will be assured that game state is consistent between turns.

Application Data Protocol Breakdown

Below are breakdowns of data that may be included in a packet. These protocols can be combined in a single data packet. Note that additional fields may be added during implementation.

Player Movement:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Type ID				Player ID				Player X-pixel location									Player Y-pixel location										Options or filled				

- Data Type ID = The Sprite code value found in Table 1.1 above.
- Player ID = The ID number for the specific worm.

- Player X-pixel location = The x location of the upper-left pixel for the Sprite on the gameboard.
- Player Y-pixel location = The y location of the upper-left pixel for the Sprite on the gameboard.
- Options or filled = This may include additional data or be stuffed.

Weapon Fired:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Type ID				Weapon ID				CF	Firing Angle								Velocity								Options or filled						
Player X-pixel location									Player Y-pixel location										Options or filled												

- Data Type ID = The Sprite code value found in Table 1.1 above.
- Weapon ID = The particular type of weapon fired.
- CF = Bit representing the direction the character was facing when weapon fired (0 is left, 1 is right).
- Firing Angle = The angle between 0 and 180° at which the weapon was fired.
- Velocity = The “strength” of the weapon fired, between 0 and 100.
- Player X-pixel location = The x location of the upper-left pixel for the Sprite on the gameboard.
- Player Y-pixel location = The y location of the upper-left pixel for the Sprite on the gameboard.
- Options or filled = This may include additional data or be stuffed.

Player Health

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Type ID				Player ID				Player Health								Options or filled															

- Data Type ID = The Sprite code value found in Table 1.1 above.
- Player ID = The ID number for the specific worm.
- Player Health = The current health for the player (typically 0 to 100, but may be altered).
- Options or filled = This may include additional data or be stuffed.

Tile Existence

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Type ID				Tile ID				Tile X-pixel location									Tile Y-pixel location										Options or filled				

- Data Type ID = The Sprite code value found in Table 1.1 above.
- Tile ID = The ID number for the type of Tile being displayed (should usually be the same as Data Type ID).
- Tile X-pixel location = The x location of the upper-left pixel for the Tile on the gameboard.

- Tile Y-pixel location = The y location of the upper-left pixel for the Tile on the gameboard.
- Options or filled = This may include additional data or be stuffed.

Reliability of Transfer

Since UDP is a connectionless transfer with only best-effort assurances, we cannot assume that all packets will arrive without error across the network. This presents two problems - one is a lost packet during transmission, the other a corrupted packet that is received.

For a packet lost in transmission, the easiest means by which to assure receipt is through a SEND-ACKNOWLEDGE protocol between the sender and receiver. Thus, when the active user sends a packet, it will await an ACK packet from the receiver acknowledging receipt of the packet. The sender will wait a specified amount of time for this acknowledgment (for example, 500 ms), and if it is not received it will timeout and resend the last packet. Multiple failures to transmit the packet will lead to termination of the game. Upon receipt of this acknowledgment from the receiver, the sender will return an ACK packet that signifies its receipt of this packet and “officially” cedes control of the game to the other computer. An identical timeout protocol to that for the earlier packet will apply.

In the event that a packet is received but is corrupted (based on an inconsistent checksum value), the receiver will not send an acknowledgment and wait for the sender to resend the packet. If corruption of a given packet occurs multiple times, the game will terminate.

V. Gameplay

Overview

A typical game will feature 2 players for each side, and victory will be achieved for a side when all players on the opposing side are dead. We may support more than 2 players per side depending on implementation and time constraints.

Game Control

The game control logic will be executed by the NIOS II RISC processor running C. The interface will consist of signal inputs for left, right, up and down arrow keys, and the spacebar. Each key must be able to output two codes: one for “key pressed,” and the other for “key released.” Since only one key may be pressed at a time, all inputs will be coded into a single integer value communicated to software, similar to the method utilized in Lab 2. The outputs of the software consist of the location of the top right pixel of each sprite and the top right pixel of each Tile, and an integer that requests a sound be played. Sprites will be 16x8 RGB arrays.

Sprite and Audio Generation

In order to simplify the game design process, two programs will be written in a high-level scripting language such as C#. The first program will convert a 256 color bitmap file into a VHDL formatted text file defining an array. This will let us design the sprites in a program such as Microsoft paint, and convert them into a format that we can copy / paste into the VHDL hardware definitions.

The second scripted program will perform a similar function for the audio files. It will take a windows waveform file (.WAV) and convert it to a VHDL-formatted array definition.
The game play will be executed as follows. First, there is an introduction screen which prompts the use for his or her team name. Each team consists of four worms, and the names of the worms are chosen randomly from a list of 40 worm name. No two worms can have the same name. After the user enters his or her name and presses enter, the program enters game mode. First, the terrain is drawn. There are three basic types of Tiles that make up the terrain (as specified in Table 1.1): ground, lava, and air. The terrain will be drawn as follows. The first Tile is always drawn at pixel 1/3 from the bottom of the screen (pixel 317) on the left hand side. After that, each Tile is placed one unit above, one unit below, or on the same level as the Tile to its left with equal probabilities. All Tiles will be rectangular. Thus, the terrain is a random walk around position 317. Next, any Tile below position 317 that is not occupied by terrain will be filled in with the lava Tile. This will ensure that roughly 50% of the terrain is lava. Lastly, any Tile that is not occupied by ground or lava will be occupied by a sky Tile.

Game Initialization and Player Interaction

Next, the worms will be placed. The worms are placed randomly, but must only be placed on “land” Tiles. Each worm consists of two sprites. The first is the worm itself, which may be facing left or right depending on which direction the worm last moved. The second is the target reticle, which starts at 45 degrees and always stays a fixed radius away from the worm. The first worm to move is chosen at random. Text will display “[Worm Name] is up from team [Team Name],” and a bobbing arrow will appear above the worm whose turn it is. Now the player starts his or her turn. The left and right arrow keys move the worm left or right while they are pressed. The up and down arrow keys change the angle of the reticle in a range of +/- 90 degrees in the direction that the worm is facing.

Once the player has moved the worm and target into position, he or she is ready to fire the projectile. A status bar will indicate the speed of the projectile from 0-100%. Pressing down the space bar initiates the fire sequence, and the status bar starts charging up from 0% to 100%. When the user releases the space bar, the speed is set for the projectile. The projectile consists of a spherical sprite that moves with constant speed in the x direction and constant acceleration in the y direction. When the projectile hits a ground Tile, several “explosion” sprites are drawn from the point of contact out to a predefined damage radius. Any worms drawn within that radius are penalized with a damage value that depends on their distance from the point of impact. If the projectile hits a lava Tile or goes off the screen, no explosion occurs. Then the first worm from the opposite team goes.

If a worm’s health decreases below 0%, the text “[Worm Name] is toast” is displayed and a “gravestone” sprite appears in its place. Players continue until all the worms on one side are eliminated. When the game is over, text displays “Team [Team Name] is victorious!” Pressing space bar automatically starts a new game.

The each explosion radius is pre-computed and stored in ram as an array, the value of each element determine the damage caused, if damage is assigned to a ground Tile, the corresponding position in the Tile matrix is set to '0'.

To determine the trajectory of the projectile the following procedure is used:

1. When the player releases the space bar two points are computed:
    a. Point T as (H,L), where H = sin(angle)*strength and L = cos(angle)* strength + wind intensity. Wind is a parameter that can be positive or negative, because of that the minimum l is fixed to '0' (the worm hurts itself). The functions sine and cosine will be implemented as a lookup tables stored in RAM.
    b. Point D as (X,Y), where X = starting x coordinate and Y = starting y coordinate + L.
    Using point T it is obtained the movement equation of the projectile: y = Ax + C. So for going towards the point T, A will be (sin(angle)*strength*2)/(cos(angle)*strength+wind). C will be '0' except when arriving to T.
2. During movement If the projectile hits anything it explodes.

After releasing the space bar, the projectile is moved smoothly towards the point T. In order to get a curved trajectory, When the projectile get closes to the point T, the parameter C starts decreasing.

When the projectile gets to T it turns to go towards D, so the A parameter will be -A. This time the parameter C will increase until be '0'. If the projectile get to D it continues with the same route (same A) until it hits land or water, in the second case it does not explode.

Game Scoring

Scoring is based on two elements: the amount of health that remains for a player as well as the number of players that remain for a given side. Each player begins with 100% health, and this will diminish based on the amount and type of damage that player receives from the opposition. Once the player's health reaches 0%, that player is dead and eliminated from that round of the game.

VI. Sound Output

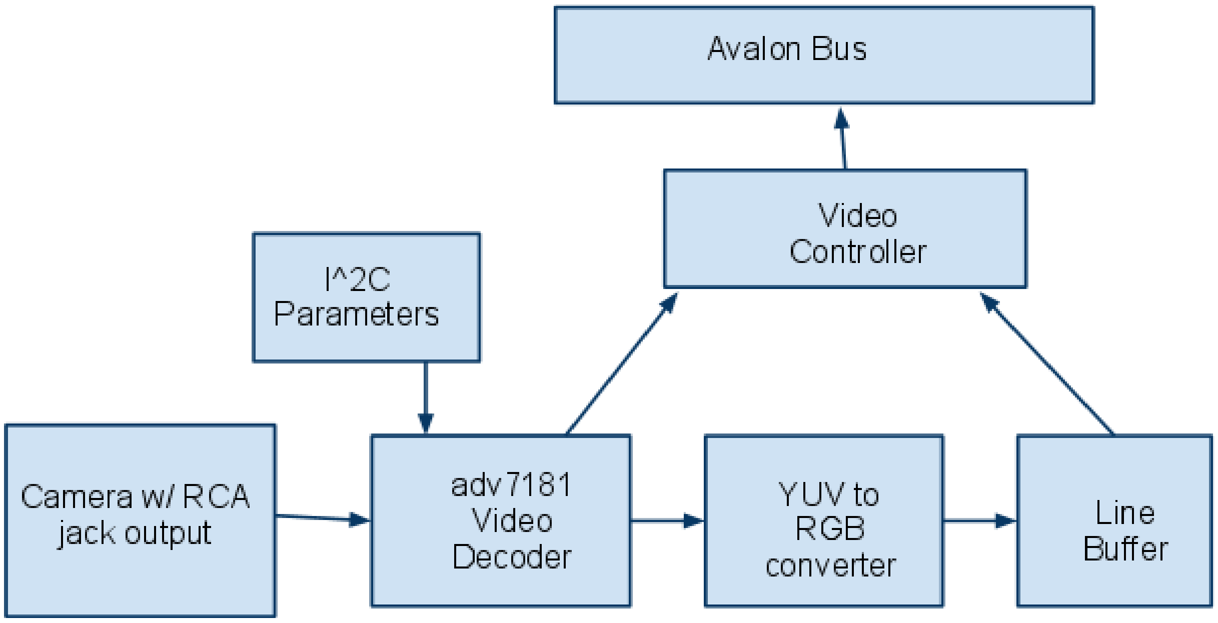
Sound will be generated using the DE2 on board WM8731 audio CODEC, as used in the lab 3. The audio CODEC will read the values of the music and sound effects from the RAM using the Avalon bus.

VII. Video Input

The Video component of this project is an element whose inclusion is optional and will be based on progress of essential elements.

To generate a "unique" background for a game session, the user will be able to utilize an attached video camera to take a "snapshot" and use that for the background of the game. The basic premise is that when the user initiates a gaming session, whatever image is currently being sent to the Video In component of the DE2 board will replace the generic background for the game. This signal will first be run through the Sobel Edge detection algorithm described below before being outputted to the screen. This will allow us to filter out extraneous noise from the video signal and improve the speed by which the background will need to be updated (since the background will be either black or white).

Video will be input to the DE2 on board ADV7181 ADC. This decoder outputs color in YUV color space so we will need to convert those colors to RGB to use in the VGA display. The color space conversion can be implemented by integer multiplication and bitshifting so it is easy to do this work in VHDL. We will buffer each line of data from the video input device and when a complete line has been input we will write the data into ram over the avalon bus. Parameters for the ADV7181 chip are set using the I2C interface so our top level module will establish that connection. Below is a more detailed block diagram of the video input hardware:



Edge Detection

Sobel Edge detection is a simple algorithm for detecting edges. The algorithm works by convolving an image with a 3 x 3 matrix approximating the horizontal, vertical or diagonal derivative. For example the convolution matrix for the horizontal derivative approximation is:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

The matrix for the vertical derivative is the transpose of the above matrix.

Since matrix multiplication is just a series of additions and multiplications and we will have the color values of the pixels stored in RAM we should be able to implement this algorithm in VHDL. For each pixel we need to locate its 8 neighboring pixels and multiply their color values by the corresponding number in the convolution matrix. The border lines are special cases as they do not have a complete set of neighboring pixels. We executed the convolution separately for each color channel and combine the results into the output color.